Monte Carlo Chess

Monte Carlo Schach
Bachelor-Thesis von Oleg Arenz aus Wiesbaden
April 2012

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Knowledge Engineering

Monte Carlo Chess
Monte Carlo Schach

Vorgelegte Bachelor-Thesis von Oleg Arenz aus Wiesbaden

1. Gutachten: Prof. Johannes Fürnkranz
2. Gutachten:

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 16. April 2012

_____

(O. Arenz)

Abstract

MCC, a UCT based Chess engine, was created in order to test the performance of Monte-Carlo Tree Search for the game of Chess. Mainly by modifications that increase the accuracy of the simulation strategy, the performance of the base implementation was improved by approximately 864 Elo. MCC performed still too bad to compete with Minimax based chess programs or to seriously suffer from search traps.

Contents

1 Introduction **4**

2 Related Work **5**
2.1 Minimax Search . . . 5
2.1.1 Alpha-Beta Pruning . . . 5
2.1.2 Iterative Deepening . . . 7
2.2 Monte-Carlo Tree Search . . . 7
2.2.1 UCT . . . 8
2.2.2 AMAF / RAVE . . . 9
2.2.3 Progressive Bias . . . 9
2.2.4 Progressive Unpruning / Widening . . . 10
2.2.5 Decisive Moves . . . 10
2.2.6 Heavy Playouts . . . 10
2.2.7 Monte-Carlo Solver . . . 10
2.3 Chess Engine Programming . . . 11

3 Prior Assessment of Monte-Carlo Tree Search for Chess **13**
3.1 Benefits . . . 13
3.2 Drawbacks . . . 13
3.3 Assessments of modifications . . . 15
3.4 Summary . . . 18

4 MCC - A Monte-Carlo Chess Engine **18**
4.1 Base implementation . . . 18
4.2 Modifications . . . 19
4.2.1 Modifications to the simulation strategy . . . 19
4.2.2 Further modifications . . . 20

5 Evaluation **22**
5.1 Base implementation . . . 22
5.1.1 Draws . . . 22
5.1.2 Accuracy . . . 24
5.2 Modifications . . . 26
5.2.1 Heavy Playouts . . . 26
5.2.2 Endgame Tablebase . . . 27
5.3 Comparison of the different modifications . . . 28

6 Conclusion **30**

7 Appendix **31**

1 Introduction

In 1997, the chess computer Deep Blue defeated the human chess world champion Garry Kasparov in a match of six games (Schaeffer 2000). Since then, chess computers became yet stronger and now exceed the performance of human players. Currently, all good computer programs are based on the tree search algorithm Minimax (Russell & Norvig 2010, p. 165).
It was also tried to use Minimax in order to create Go programs, however, Minimax did not yield similar good results in Go as in Chess (Gelly et al. 2012). In 2006, a new tree search algorithm was proposed, Monte-Carlo Tree Search (Coulom 2006, Kocsis & Szepesvári 2006, Chaslot et al. 2006), which was successfully used to create strong Go programs (Wang & Gelly 2007). Monte-Carlo Tree Search (MCTS) performs better than Minimax in the game of Go. Does it also perform better in the game of Chess?
Although there are indicators that MCTS is not suited for the game of Chess (Ramanujan et al. 2010), this thesis tests its performance with respect to Chess and examines how modifications to the base algorithm can improve it. For this purpose, a Chess engine based on Monte-Carlo Tree Search is created and compared with several modifications. Some problems encountered during these tests are illustrated on concrete chess positions. The algebraic notation[1] is used to describe the chess moves on these positions.
The remainder of this thesis is structured as follows:
Section 2 outlines previous related work. The Minimax algorithm is covered in Section 2.1, since it is the main competitor for Monte-Carlo Tree Search in the game of Chess. Understanding the currently best-performing algorithm helps to understand the advantages and disadvantages of MCTS. Section 2.2 covers Monte-Carlo Tree Search - the algorithm which is of main interest of this thesis. Additionally, several modifications of the base algorithm are explained. These modifications led to significant improvements on different games. Section 2.3 deals with some techniques that are currently used by chess engines and do not depend on the underlying algorithm.
In Section 3 the prospects of MCTS for Chess are analyzed. For this purpose, the algorithm is compared to Minimax search and the prospects of the different modifications are assessed with respect to Chess.
Section 4 discusses the implementation of the MCTS Chess program MCC which was created to further analyze the performance of Monte-Carlo Tree Search for the game of Chess. Several versions of MCC were created in order to test the different modifications independently.
The results of these tests are presented and analyzed in Section 5.
Finally, Section 6 recaps the considerations and insights provided by this thesis.

[1] see http://www.fide.com/component/handbook/?id=125&view=article

2 Related Work

2.1 Minimax Search

Assuming best play by both players, zero-sum two-player games like Chess can be labeled either as winning for player A, winning for player B or draw. However, if the game complexity is too large, finding the correct label is not feasible. Therefore, heuristic evaluation functions are used to assign a numeric value to a given game state. For instance, a positive value could indicate that player A is assumed to be winning and a negative value could indicate that player B is assumed to be winning. High absolute values indicate high confidence in the assumption. Such evaluation functions can be created by using domain knowledge to evaluate different features of the game state.
Features commonly used by Chess evaluation functions include:

- Material balance
- Piece activity
- King safety
- Pawn structure

A simple evaluation function would now calculate the weighted sum of the scores of each feature. Chess evaluation functions usually normalize the resulting score to a unit of measure called centipawn, where an advantage of 100 centipawns should correlate to the advantage of having an extra pawn in the middlegame. Dynamic possibilities have to be considered as well, e.g. a player might be able to reach a better game state by force. These dynamic aspects can hardly be identified by only examining static positional features.
Minimax Search (Russell & Norvig 2010, p. 165) is a method that deals with the dynamics of a game state by applying the evaluation function to all states which can be reached after a given number of moves, instead of directly applying it to the current game state. Therefore, a game tree with a fixed depth is created and the evaluation scores are stored for each leaf node. Then the score of the leaf nodes are propagated upwards layer-wise until a score gets assigned to the root node. Depending on which player is to move, either the maximum or the minimum score of its children is propagated to an inner node. The player that wants to reach a high evaluation score is the maximizing player and the player that wants to reach a low evaluation score is the minimizing player. Figure 1 shows the result of a Minimax Search with depth 2. Initially scores were only assigned to the leaf nodes. As each node in the middle layer belongs to the minimizing player, the lowest score of its children was used to evaluate it. Finally, the highest score in the middle layer was propagated to the root node.
Instead of propagating the values layer-wise, it is also possible to propagate them in a depth-first manner. This approach has better memory-efficiency and allows for a crucial modification called Alpha-Beta Pruning.

2.1.1 Alpha-Beta Pruning

Minimax search with Alpha-Beta pruning (Knuth & Moore 1975), also referred to as Alpha-Beta search, leads to the same root node evaluation as Minimax search but does not necessarily need to evaluate all leaf nodes (Russell & Norvig 2010, pp. 167-169). The basic idea of Alpha-Beta

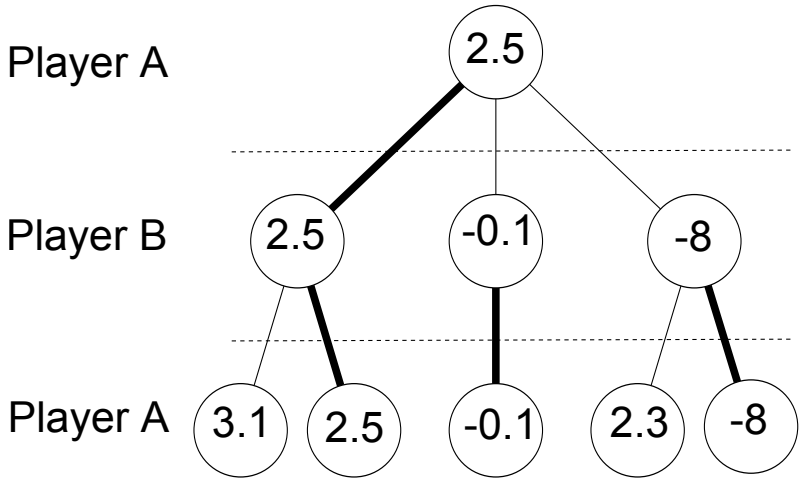

Figure 1: Minimax applied to a game tree with depth 2. Player A is maximizing, player B is minimizing.

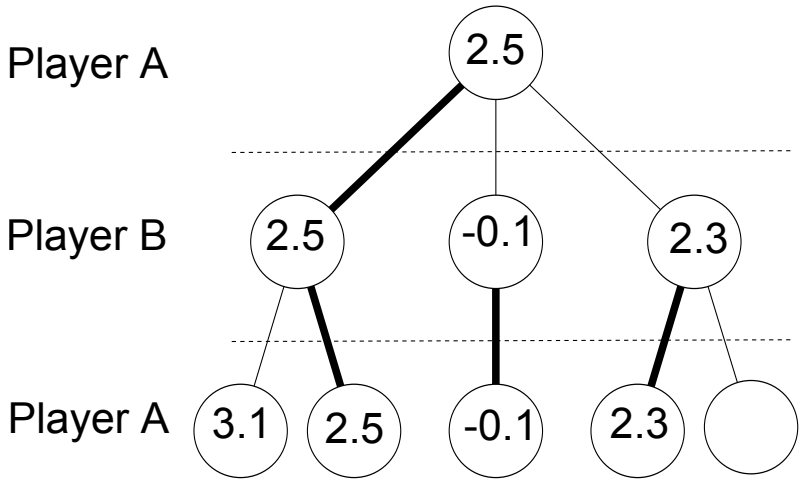

Figure 2: Alpha-Beta applied to the same game tree as in Figure 1. The right-most leaf node is not evaluated due to an α-cutoff. The value of its parent is larger than its true minimax value, but the root node evaluation can not be affected.

pruning is as follows: Assuming that, when examining an inner node, the minimax value v_i of a move m_i was already discovered. If, by examining a different move m_j of that node a reply is found that gives the opponent a better value than v_i, m_j can be discarded immediately without evaluating other replies. In order to discover the minimax value of a child without evaluating the leaf nodes that belong to other children, Alpha-Beta search needs to update the values in a depth-first manner. It also uses two extra variables: α stores the maximum score the maximizing player is proven to reach and β stores the minimum score the minimizing player is proven to reach, respectively. Initially α is set to $-\infty$ and β is set to $+\infty$. Whenever during the search the minimax value of a node belonging to the minimizing player is discovered and is lesser or equal α, the evaluation of its parent node is aborted, because the maximizing player would not choose that node - an α-cutoff occurred. Respectively, a β-cutoff occurs if the minimax value of a node that belongs to the maximizing player is greater or equal β. Figure 2 shows the result

of an alpha-beta search with a left-to-right depth-first search on the same game tree as in Figure 1. The right-most leaf node is not evaluated as its score can not affect the minimax value of the root node. It is to note that good move-ordering leads to more cut-offs and cut-offs may lead to wrong minimax values for inner nodes.

2.1.2 Iterative Deepening

Minimax search uses a game tree with fixed depth and can only provide a meaningful result after the search is completed. In order to make better use of the available game time, it is usually combined with Iterative Deepening (Korf 1985) in the game of Chess. Iterative Deepening starts with a low depth Minimax search and increments the depth until the algorithm is stopped. Carrying out several iterations of Minimax search in that manner does not produce much overhead because the number of nodes in the game tree grows exponentially with the depth. It is also to note that previous iterations are usually not wasted since their results can be used to reduce the time needed for the next iteration, e.g. by move ordering in combination with Alpha-Beta search.

2.2 Monte-Carlo Tree Search

Monte-Carlo methods (Meteopolis & Ulam 1949) try to estimate the solution of a given problem by aggregating the results of random simulations. They are often applied to problems that are infeasible to solve by deterministic algorithms. Following the same idea, random simulations can be used to estimate the outcome of a given game state. For instance, a heuristic evaluation function for chess positions can be constructed by performing a large number of random playouts from the given position and counting the number of wins for White and the number of draws. The evaluation score can now be calculated:

$$\text{score} = \frac{\text{total reward}}{\text{number of simulations}}$$

$$\text{total reward} = \text{number of white wins} + 0.5 \times \text{number of draws}$$

Unlike the static evaluation function discussed in section 2.1 the simulation-based heuristic produces a score that correlates to a winning probability. To be specific, the evaluated score converges with increasing simulations to the winning probability of the white player, *if both players chose their moves at random*. As Chess is a zero-sum game the winning probability of the black player is the converse winning probability of the white player. Similar to an evaluation function that is based on static features of a position, the simulation based heuristic fails to identify the dynamic aspects of the game. For example, on positions where the side to move has only few moves that are winning and many moves that are losing, most of the simulations would start with a losing move resulting in low winning probabilities for the side to move, although it has a winning position. Again, the inability of the heuristic to identify the dynamic aspects of a position can be solved by combining it with a tree search.
Monte-Carlo Tree Search (Coulom 2006, Kocsis & Szepesvári 2006, Chaslot et al. 2006) combines the simulation based heuristic with a minimax search on a gradually expanding game tree. Initially, only the root node of the game tree - representing the current board position - is created. Then the game tree grows by repeatedly performing the following four steps:

1. Select: The game tree is traversed until a node is reached which is not fully expanded yet. The branches are chosen according to a selection strategy. The selection strategy must ensure that promising branches (i.e. those branches which led to the best results so far) are picked more often than less promising branches. This leads to a best-first search, which is necessary to allow convergence of the winning probability of a node towards the winning probability of its best child. Depending on the side to move the selection strategy prefers the move with either the maximum or the minimum winning probability, similar to minimax search. However, the selection strategy must also prefer branches which have been visited less often because their estimated winning probabilities have greater variance.

2. Expand: The game tree gets expanded by adding a new child to the selected node. The child represents the resulting position, after making a legal move which was not represented in the game tree yet.

3. Simulate: Beginning from the position of the newly created node the simulation plays random moves until a terminal game state is reached.

4. Update: For all nodes which have been visited in this iteration, including the new node, the visit counter is incremented and the total reward is increased depending on the outcome of the simulation.

The four steps of MCTS are illustrated in Figure 3. For better illustration, the rules of the underlying game should allow exactly two moves for every possible game state. Hence the selection step stops on the first node which has less than two children.

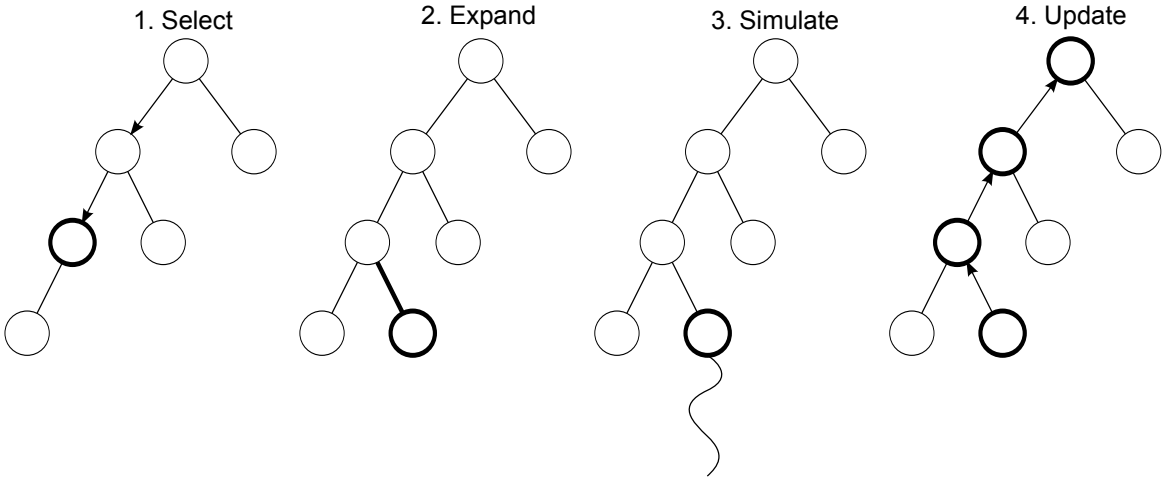

Figure 3: The four steps of Monte-Carlo Tree Seach illustrated on a game with a constant branching factor of 2.

2.2.1 UCT

A selection strategy which chooses the most promising branch often enough to ensure convergence without neglecting the other branches is essential for Monte-Carlo Tree Search. The problem of finding a good trade-off between choosing the empirical best action and exploring other actions is known as the exploitation-exploration dilemma in the multi-armed bandit

problem (Auer et al. 1995). UCT (Kocsis & Szepesvári 2006) is rather a realization than a modification of Monte-Carlo Tree Search because it provides the necessary selection strategy. UCT uses the bandit algorithm UCB1 (Auer et al. 2002) as a selection strategy:

$$i_{chosen} = \underset{i \in 1,\dots,K}{\arg\max}(w_i + C * \sqrt{\frac{2 \ln n}{n_i}}) \tag{1}$$

where w_i is the empirical winning probability of child i, n is the number of visits of the current node, n_i is the number of visits of child i, K is the number of children of the current node and i_{chosen} is the chosen child. The exploration coefficient C is by default set to 1 and can be used to steer the strategy towards more exploration ($C > 1$) or towards more exploitation ($C < 1$). UCT is not the best selection strategy (Tesauro et al. 2010), but it is simple and was proven correct. Therefore, it is often chosen for MCTS implementations.

2.2.2 AMAF / RAVE

Instead of using the result of a simulation only to update those nodes which were visited in the selection or expansion step, the AMAF (Gelly & Silver 2007) modification additionally updates some siblings of these nodes. A sibling is updated, if it could be reached by making a move from its parent node that was played during the simulation by the corresponding side to move.
Gelly & Silver (2007) created the RAVE (Rapid Action Value Evaluation) heuristic by applying AMAF to UCT. The updates to the siblings are less meaningful than the updates to the visited nodes, because during the play-out the corresponding moves were played on a different game state. Thus, the AMAF updates are maintained separately in RAVE. The RAVE value can now be used mainly for sparsely explored nodes by adjusting the selection strategy:

$$i_{chosen} = \underset{i \in 1,\dots,K}{\arg\max}(\beta_i * w_{RAVE,i} + (1 - \beta_i) * w_i + C * \sqrt{\frac{2 \ln n}{n_i}}) \tag{2}$$

where $w_{RAVE,i}$ is the RAVE value of child i and β_i is a weighting coefficient, that depends on the statistics of child i. Gelly & Silver (2011) used the following formula to calculate β_i:

$$\beta_i = \sqrt{\frac{k_{RAVE}}{3n_i + k_{RAVE}}} \tag{3}$$

where k_{RAVE} can be set to the number of simulations at which β should be $\frac{1}{2}$.

2.2.3 Progressive Bias

Guiding the selection for nodes with only few visits can also be achieved by using static heuristic functions like the one discussed in Section 2.1. Like in RAVE, the bias to the selection strategy should decay with increasing simulations in order to allow the selection strategy to make better use of the accumulated statistics. Progressive Bias (Chaslot et al. 2008) is a modification of the selection strategy that incorporates a heuristic function $H(n)$ with progressively decaying influence:

$$i_{chosen} = \underset{i \in 1,\dots,K}{\arg\max}(w_i + C * \sqrt{\frac{2 \ln n}{n_i}} + \frac{k_{bias} * H(n_i)}{n_i}) \tag{4}$$

where k_{bias} is set suitable to the heuristic function.

2.2.4 Progressive Unpruning / Widening

Progressive Unpruning (Chaslot et al. 2008) and Progressive Widening (Coulom 2007) are two very similar approaches which were invented independently. They soft prune children with bad heuristic scores in the early phase of exploring a node and progressively "unprune" them when the node gets better explored. Soft pruning is achieved by ignoring them in the selection step. The nodes are unpruned in the order of their heuristic value. The number of unpruned nodes usually grows logarithmically with the number of simulations. Progressive Unpruning prunes the children except a fixed number of them as soon as the node is created. Progressive Widening applies pruning to a node only after fixed number of simulations.

2.2.5 Decisive Moves

Apart from modifying the selection strategy, the performance of Monte-Carlo Tree Search can also be improved by modifying the simulation strategy. Teytaud & Teytaud (2010) examined the performance of a simulation strategy which always plays a direct winning move, if such a move is available. They showed that, although additional time had to be spent for identifying winning moves, their simulation strategy performed better than a purely random strategy in the game of Havannah. Additionally, identifying anti-decisive moves (i.e. moves that prevent the opponent from playing a decisive move) was demonstrated to lead to further improvements (Teytaud & Teytaud 2010).

2.2.6 Heavy Playouts

The simulation strategy can also be improved by utilizing domain knowledge. This is usually connected with a trade-off between simulation speed and simulation accuracy. Within this thesis, high accuracy of a simulation strategy should be understood as the ability of the strategy to produce expected values that for most nodes correlate to the expected values produced by a perfect playing strategy. Heavy playouts (Drake & Uurtamo 2007) use domain knowledge to create more accurate simulations at the cost of simulation speed. There are many ways to create heavy playouts. For instance, Wang & Gelly (2007) use patterns to produce sequence-like random simulations with great success for their Go program MoGo. The patterns are used to find interesting moves around the last played stone. If interesting moves are found, one of them is randomly chosen and played. Only if no pattern matches, a completely random move is played. Another form of heavy playouts was used by Winands & Björnsson (2009) for the game of Lines of Action. They used a heuristic function to create a mixed simulation strategy which consists of two different strategies. The simulation begins with the corrective strategy which chooses a move with a probability according to its heuristic evaluation score. Moves that are evaluated worse than the current position get their score artificially reduced to a fixed minimum, which is close to zero. After a certain amount of moves the simulation switches to the greedy strategy. The greedy strategy always plays the move with the best heuristic evaluation. Both strategies stop as soon as a move is found that has a heuristic score above a certain threshold. In that case, the simulation considers the corresponding side to move as winning. In their experiments the mixed strategy performed better than the greedy strategy and the corrective strategy.

2.2.7 Monte-Carlo Solver

Monte-Carlo Solver (Winands et al. 2008) is used to prove the game-theoretical value of a node if such a prove is feasible. The game-theoretical value of a leaf node is obviously already proven

if it corresponds to a terminal game state. Monte-Carlo Solver only proves game-theoretical wins and losses. A node is proven winning if at least *one* of its children are proven losing. A node is proven losing if *all* of its children are proven winning. A node gets its rewards assigned to $+\infty$ or $-\infty$ if it is proven winning or losing, respectively. The update mechanism is modified to deduce the game-theoretical value of a node from the rewards of its children, if such a deduction is possible. This modification also led to an improvement of playing strength in the game Lines of Actions due to faster convergence to game theoretical values (Winands et al. 2008).

2.3 Chess Engine Programming

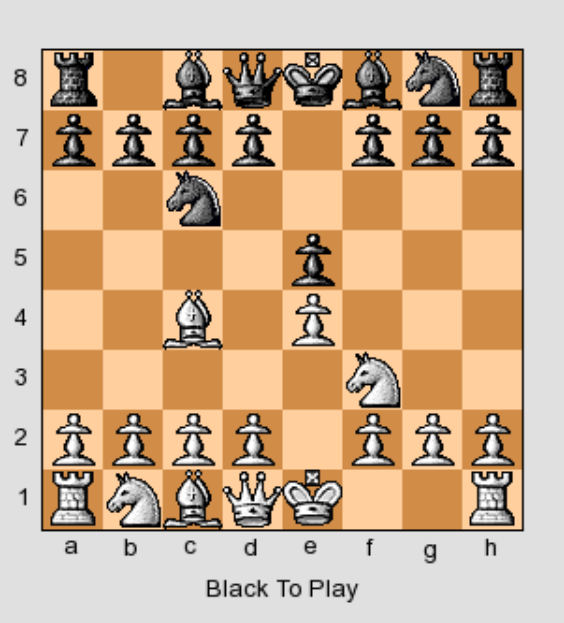

Figure 4: Different move sequences can lead to the same chess position.

Current state-of-the-art chess programs also apply several techniques that are not specific to the underlying Iterative Deepening Alpha-Beta algorithm. As some of these led to critical improvements in playing strength, they should be considered for MCTS based chess players as well.

- Bitboards: A bitboard (Adel'son-Vel'skii et al. 1970) is a data structure that is commonly used for representing chess positions. Bitboards use only a single bit for every square of the board. Hence, for every piece type and every color an own bitboard is required. Additional bitboards are often used, e.g. in order to mark blocked squares or attacked squares. This data representation allows move generation by using binary operations on the bitboards which is significantly faster than using nested loops.

- Transposition Tables: In the game of Chess, different move sequences can lead to the same chess position. For instance, 1.e4 e5 2.Nf3 Nc6 3.Bc4, 1.e4 e5 2.Bc4 Nc6 3.Nf3, 1.e3 e6 2.e4 e5 3.Nf3 Nc6 4.Bc4 and many other move sequences lead to the same position as illustrated in Figure 4. Different move sequences which lead to the same position are called transpositions in Chess. By sharing the evaluation score of a node with its transpositions significantly less nodes have to be evaluated. Alpha-beta based chess programs usually

use transposition tables (Breuker et al. 1977) to share information among transpositions - most notably the program MackHack by Greenblatt et al. (1967) which was the first chess program to use transposition tables. A hash function is used to calculate the index of the corresponding table entry for a position.

- Endgame Tablebases: Chess endgames with only few pieces left can be solved by a retrograde analysis beginning with all possible mating positions for a given material constellation (Bellman 1965). The results of the analyses of every material constellation with N pieces on the board can be used to create a N-piece endgame tablebase. Such tablebases can be queried to quickly obtain the correct evaluation of all chess positions with N pieces.
- Static Exchange Evaluator: If a piece is attacked and defended multiple times, the side to move has to decide whether it is in its interest to capture the piece or not. Regarding only the material implications, the least valuable attacker should always be used for a capture. A Static Exchange Evaluator (SEE) is an algorithm that examines all sequences of captures on a given square (Maynard Smith & Michie 1961). It determines the material gain or loss for the side to move assuming that both sides make the best decisions regarding only those captures on the given square. The Static Exchange Evaluator can also be used on non-capturing moves. In that case the returned score is either negative or zero.

3 Prior Assessment of Monte-Carlo Tree Search for Chess

3.1 Benefits

Monte-Carlo Tree Search has three main benefits:

- MCTS does not depend on a heuristic evaluation function. Hence, besides the game rules which need to be known for move generation and evaluation of terminal game states, no domain knowledge is required to construct a MCTS based game player. This makes the algorithm particularly interesting for general game playing and games for which insufficient domain knowledge is available to construct good heuristic functions.
- MCTS is anytime. Whenever the algorithm gets stopped, it can suggest a move by making use of all completed iterations. This is important because game playing programs usually have limited time to choose a move.
- MCTS is best-first. Monte-Carlo Tree Search is a best-first search because the most promising nodes are most explored - the game tree grows asymmetrically. It is desirable to spend most time in evaluating promising moves, as they are most important for evaluating the current position.

Assessing the usefulness of these benefits with respect to Chess it is first to note that although Monte-Carlo Tree Search does not need heuristic evaluation functions, incorporating domain knowledge may lead to significant improvements, e.g. by Heavy Playouts. Furthermore, current Alpha-Beta based chess engines demonstrate that heuristic evaluation functions can be used to beat even the best human chess players. As good heuristics exist and would probably be used anyway by strong MCTS based chess programs, the benefit of not requiring them is negligible.
The anytime property of Monte-Carlo Tree Search is more likely to be useful for chess programs. Although, Alpha-Beta search with Iterative Deepening makes also use of all completed iterations, the time which is needed per iteration grows exponentially. However, this benefit is probably not telling as well because chess players can usually ration their time. Alpha-Beta based players therefore prefer to stop the search before starting a new iteration, if that iteration is unlikely to be completed due to time limitations.
Traversing the game tree in best-first order is the most promising benefit of Monte-Carlo Tree Search with respect to Chess. Whether the simulation strategy leads to the desired asymmetrical tree structure depends on the consistency of the strategy. Within this thesis, high consistency of a simulation strategy should be understood as the ability of the strategy to converge to approximately equal winning probabilities for any node and its best child. A simulation strategy with high accuracy has also high consistency. Whether random or pseudo-random simulations are consistent in Chess has yet to be determined.

3.2 Drawbacks

The MCTS approach also has potential drawbacks compared to Minimax search:

- Search traps occur if a player is able to improve his position by a combination of moves that includes at least one seemingly bad move. An example of such a search trap in Chess is given in Figure 5: The move 1.Nxe5 utilizes a well-know chess trap, which is called

the Légal Trap. By playing that move White allows Black to capture the white queen by 1...Bxd1, which gives Black a large advantage in material. Therefore, 1.Nxe5 might look like a very bad move at first glance. However, after finding out that Black gets mated by force, if White continues with 2.Bxf7+ Ke7 3.Nd5#, it becomes obvious that Black should not capture the queen and instead accept the loss of a pawn by playing 1...dxe5 2.Qxg4. This example highlights an aspect of Chess which is important for the evaluation of the best-first search of MCTS: Some good moves seem very bad until a specific continuation is discovered. The influence of search traps for MCTS has already been examined for Chess by Ramanujan et al. (2010). They discovered that shallow search traps occur often in the game of Chess and that UCT needs significantly more iterations to identify them than Minimax search. Their results can be explained by the nature of Monte-Carlo Tree Search: Best-first search spends little computation time for evaluating the trap move as long as the concrete winning continuation is not fully discovered. As only little time is spent for exploring the trap move, many iterations are required to discover the correct continuation - a vicious circle. In contrast to MCTS, Minimax search with Iterative Deepening is a breadth first search and is therefore able to identify search traps much quicker.

Figure 5: Example of a search trap.

- Move generation speed is a decisive factor in Monte-Carlo Tree Search, as - in contrast to Minimax search - it is not only required for creating the game tree but also during simulation. MCTS was used successfully for creating Go programs. However, in the game of Go move generation is almost as simple as finding empty intersection, whereas it is more complicated in the game of Chess. Chess pieces are actually moved. Therefore, not only the target square has to be considered but also the initial square of the piece as well as all squares that are traversed (except for knight moves). Furthermore, Chess has six different piece types which move all differently. Especially pawn moves are quite intricate because they do not capture in the same way as they move. Additionally, en-passant captures, castling and pawn double steps are exceptional moves which are only legal under certain

conditions. Finally it must be checked for every move whether it leaves the own king in check.
While, by utilizing Bitboards, move generation turned out fast enough to enable Minimax based players to perform very well in Chess, it is an open question whether it is fast enough to enable MCTS based players to perform similarly well.

- The accuracy of random simulations is another important factor that influences the performance of Monte-Carlo Tree Search. The winning probability of a random or pseudo-random strategy has usually a different expected value than the winning probability of a perfect playing strategy. Technically, even with the worst simulation strategy MCTS would still converge to the correct evaluation because the simulation strategy becomes irrelevant as soon as the game tree is fully expanded. However, because it is usually infeasible to fully expand the game tree, a simulation strategy with low accuracy is likely to lead to bad evaluations. It should be noted that MCTS is able to compensate for several nodes which have inaccurate expected values, because only one simulation is started per node. Nevertheless a certain degree of accuracy is required in order to produce reasonable evaluations within limited time. Whether random simulations have sufficient accuracy in the game of Chess has yet to be investigated.

3.3 Assessments of modifications

The performance of MCTS based game players is usually significantly increased by applying several modifications to the base algorithm. Therefore, when assessing the prospects of MCTS for Chess these modifications should not be ignored.

- AMAF and especially RAVE proved useful in the game of Go. In Go, many moves that are played during the simulation are also available at the start of the simulation. In the game of Chess, the pieces are moved around during the simulation. Hence, they occupy different squares and have different moves available. Consequently, Chess does not suit well for RAVE and similar AMAF modifications. Whether RAVE estimates in Chess are good enough to provide verifiable improvements is also questionable. However, such assessments seem unnecessary as, for the reason stated earlier, RAVE is most likely no significant improvement anyway.

- Progressive Bias, Progressive Widening and Progressive Unpruning use heuristic evaluation functions to guide the selection strategy for nodes that have not been visited often enough to provide meaningful winning probabilities. As heuristic functions proved very powerful for Minimax players in Chess, it is interesting to combine them with MCTS as well. Biasing the selection strategy towards promising nodes accelerates the convergence of winning probabilities. For these reasons, Progressive Bias, Progressive Widening and Progressive Unpruning are promising modifications with respect to Chess.

- Identifying decisive moves in order to always play them during the simulation paid of in the game of Havannah (Teytaud & Teytaud 2010). It was already noted that move generation in Chess is expensive compared to many other games. If the simulation strategy is able to generate a random move, without needing to generate all possible moves, identifying decisive moves might become too costly. However, if the simulation strategy generates all moves anyway, identifying decisive moves produces little overhead, because checking the

resulting positions for mate can be done fast by using Bitboards. Decisive Moves might then even be faster than using a purely random strategy due to shorter simulations. Additionally, shorter simulations should generally improve the simulation accuracy. If no extra move generations are required Decisive Moves is a promising modification with respect to Chess.

- Heavy playouts turned out to be vital for many MCTS based game player, e.g. MoGo (Wang & Gelly 2007). Sequence-like random simulations similar to those used by MoGo are promising for Chess as well. By looking for good replies to the previous move, the simulation strategy might become much more accurate and consistent. Although patterns can also be used in Chess, different approaches might be suited better. Creating sequence-like moves in Chess is however little explored, if at all.
  It seems again reasonable to make use of the fact, that strong heuristic functions are available for the game of Chess. Winands & Björnsson (2009) showed that simulations based on evaluation functions can increase the performance of MCTS. It might be even a better approach to use heuristic functions that are able to directly suggest good candidate moves. This would be more efficient than evaluating all positions which arise after making a move and then choosing a move that leads to a position with a good evaluation.
  Improving the accuracy of simulations was vital for other games and is probably vital for Chess as well. It is, however, an open question how the accuracy should be improved with respect to Chess.

- Although Monte-Carlo Solver led to an improvement in the game Lines of Actions (Winands et al. 2008) it should be primarily seen as a modification that aims at proving game-theoretical values. Whether such proofs can be used for significant improvements depends on the concrete implementation of the game player. It should be stressed, that Monte-Carlo Solver is only able to prove a forced mate, if all variations that lead to a mate have already been discovered. MCTS converges very quickly towards the correct winning probabilities, once the relevant variations for proving a win or a loss are discovered. The speed gained by marking such nodes as winning or losing and accounting for these marks in the selection strategy is therefore negligible. It is to note, that the difficulty of identifying search traps remains. Nevertheless, it is convenient to be able to prove a forced mate in the game of Chess as more meaningful output is possible. It should also be easy to calculate the mating distance by modifying the update mechanism in order to additionally count the distance to mate. Monte-Carlo Solver is an interesting modification with respect to Chess, because it can provide for more meaningful output.

Some techniques that are often used in state-of-the-art Minimax chess programs were described in Section 2.3. As these techniques do not require Minimax search, it is tempting to use them for MCTS based chess players as well.

- Bitboards are used by Minimax based chess players to quickly identify features of a chess position, most notably the available moves. As fast move generation is vital for Monte-Carlo Tree Search, it makes sense to utilize Bitboards for MCTS based chess programs. It should be emphasized, however, that MCTS only requires a single random move for the simulation strategy, whereas Minimax has to generate all legal moves. This could be exploited by first generating only pseudo-legal moves, i.e. moves which are not tested against whether they leave the own king in check. Then one of the pseudo-legal moves could be randomly chosen and tested. If the move is legal, it can be played immediately. If

it is not legal, it is removed from the set of pseudo-legal moves and the algorithm repeats by choosing another pseudo-legal move. It would be even faster to randomly choose a piece and generate only the moves of that specific piece. This approach, however, would not choose every move with the same probability, which could affect the accuracy of the simulation.

- Transpositions are very common in the game of Chess. As the same chess position can be reached by different move sequences, several different nodes in the MCTS game tree correspond to the same position without sharing any information. Saffidine et al. (2011) showed that sharing information between those nodes by building a directed acyclic graph instead of a tree can lead to improvements for several different games.

- Endgame tablebases make it possible to quickly obtain the correct evaluation for chess positions with only limited number of pieces. Minimax based chess programs can use an endgame tablebase to deduce perfect play for such positions, but they can hardy make use of it, if too many pieces are still on the board. MCTS, however, can query the tablebase not only for endgame positions within the game tree but also for endgame positions which are encountered during simulations. Endgame tablebases can therefore shorten the simulations which leads to an increased accuracy. If the time spent for querying the tablebase is less than the time saved by shorter simulations the use of endgame tablebases can additionally increase the simulation speed. Consequently, endgame tablebases might be beneficial for MCTS not only when evaluating endgame positions but already in the very beginning of a chess game.

- A Static Exchange Evaluator might also be useful for Monte-Carlo based chess programs, e.g. by replacing the heuristic evaluation function for Heavy Playouts. In the simulation strategy many blunders can be avoided by avoiding moves with negative SEE values. By preferring moves with positive SEE values bad moves of the opponent are also punished more often. This might increase the accuracy of the simulation strategy. It is to note, that a Static Exchange Evaluator is usually considerably faster than a heuristic evaluation function[2].
  Evaluation functions that include several features of a position are more suitable for creating strong chess programs than evaluation functions that only consider material balance. For that reason, chess programs use the Static Exchange Evaluator only to assist the feature-based heuristic evaluation function (e.g. for move ordering) which is responsible for the actual evaluation of the leaf nodes in the Minimax game tree. For the same reason, a simulation strategy that choses its moves according to a heuristic evaluation function might play stronger than a simulation strategy based on SEE values. However, it should not be wrongly concluded that the stronger strategy is also more accurate. Experiments by Gelly & Silver (2007) showed that an objectively stronger simulation strategy may lead to worse performance in UCT even when the speed of the strategies are comparable. Silver & Tesauro (2009) identified the balance of a simulation strategy as the correct measurement for its accuracy. They argue that a badly playing strategy would still lead to the correct evaluation, if the errors of one player are compensated by the errors of the other player.
  Therefore, using a Static Exchange Evaluator instead of the heuristic evaluation function might not only increase the simulation speed but also the simulation accuracy.

[2] like the one sketched in section 2.1

3.4 Summary

Best-first search is the main benefit of MCTS compared to Minimax search with respect to Chess. The inability to identify search traps is, however, a consequence of this search order and might hinder the creation of a strong chess program based on MCTS.
Several modifications to MCTS that were successfully used for other games are promising for Chess, too. Especially modifications that use heuristic evaluation functions are well suited for MCTS based Chess programs. The only unpromising modification assessed within this thesis is AMAF/RAVE. Algorithm independent techniques that are used with great success in top rated chess programs are also promising for MCTS based Chess programs. Monte-Carlo Tree Search might even be able to make better use of Endgame Tablebases than Minimax search.
The decisive factors for the performance of the algorithm are the accuracy of the simulation strategy as well as the speed of move generation.

4 MCC - A Monte-Carlo Chess Engine

4.1 Base implementation

We implemented a Monte-Carlo based chess engine, MCC, in order to test the performance of Monte-Carlo Tree Search for Chess. MCC is based on the strong open source Alpha-Beta chess engine Stockfish[3], which is written in C++. By using one of the strongest available chess engines, we could make use of its fast move generation based on Bitboards, its heuristic evaluation function, its Static Exchange Evaluator as well as other sophisticated implementations. In *uci.cpp*, we replaced the call to the Iterative Deepening Alpha-Beta search by a call to our UCT search which was implemented in *uctSearch.cpp*. The pseudo-codes of the most important functions can be found in the Appendix. The structure of the main function in *uctSearch.cpp* is given in Algorithm 7.1. A very simple time management is used by assigning a fixed fraction of the remaining clock time for deciding on the next move. We do not store positions within the game tree for memory efficiency.
The four steps of Monte-Carlo Tree Search were implemented in the class that represents the tree nodes, *montecarlotreenode.cpp*. The pseudo-code of our implementation of the selection step is given in Algorithm 7.2. We used the UCT formula as described in Formula 1 in Section 2.2.1.
Our implementation of the expansion step is shown in Algorithm 7.3. The moves are expanded in the same order as they are generated.
Algorithm 7.4 shows our simulation strategy. When assessing the usefulness of Bitboards in Section 3.3 it was noted that generating all legal moves is not the most efficient way, if only one random move is required. However, we decided to generate all moves nevertheless by using Stockfish's Move Generator, because this is less error-prone. This work focuses on the general prospects of Monte-Carlo Tree Search applied to Chess, hence improvements which only increase the speed of iterations are less significant for this thesis. The impact of such modification can always be estimated by providing extra evaluation time.
Our update procedure is straightforward and uses recursion. For the sake of completeness, it is illustrated in Algorithm 7.5.

[3] obtainable from: http://www.stockfishchess.com/

4.2 Modifications

We also implemented several modifications to the base algorithm in order to test their performance in respect to Chess. Before explaining these modifications we should give our reasons for not implementing some of the discussed modifications:

- AMAF/RAVE was assessed as unpromising for Chess in Section 3.3. Extracting additional information from the simulations is an interesting idea and AMAF based modifications might very well lead to some improvements in the game of Chess, too. However, for the reasons given earlier, it is rather unlikely that these improvements are deciding.
- Even so Progressive Widening and Progressive Unpruning are interesting modifications in respect to Chess, we did not test them as they are similar to Progressive Bias. We preferred to test Progressive Bias, because its implementation is less time-consuming.
- We did also not make use of transpositions. Sharing information between transpositions would certainly improve the performance of MCC, if done in a sensible way. However, identifying transpositions and handling them correctly involves considerable changes to the source code. Additionally, such modifications have the same effects as increasing the speed of iterations.

4.2.1 Modifications to the simulation strategy

The assessment has shown that the accuracy of the simulation strategy is the deciding factor for the performance of MCTS for Chess. We test three different ways to increase the accuracy of the simulation strategy: Heavy Playouts, Endgame Tablebase and Decisive Moves.

- **Heavy Playouts:** There are many ways to create Heavy Playouts. The pattern-based approach by Wang & Gelly (2007) and the mixed strategy by Winands & Björnsson (2009) were already described in Section 2.2.6. We use an ϵ-greedy policy (Sutton 1995) by playing the best evaluated move with probability $1 - \epsilon$ and a random move with probability ϵ. For this purpose, we created two modifications to our chess program: $MCC\_HP_{EVAL,\epsilon}$ uses Stockfish's heuristic evaluation function to identify the best move. $MCC\_HP_{SEE,\epsilon}$ uses Stockfish's Static Exchange Evaluator. With $\epsilon = 1$ both modifications behave exactly like unmodified MCC; with $\epsilon = 0$ both modifications are completely deterministic. Gelly & Silver (2007) showed that the performance of a simulation strategy may drop if it becomes too deterministic.
- **Endgame Tablebase:** How Endgame Tablebases might be useful to increase the accuracy of a simulation strategy was already outlined in Section 2.3. We use the open source Gaviota Endgame Tablebase[4] to stop the simulations as soon as the Endgame Tablebase can be queried. This modification uses *tb-probe.cpp* as an adapter to call the Gaviota Endgame Tablebase API.
- **Decisive Moves:** Teytaud & Teytaud (2010) demonstrated that identifying decisive moves paid of in the game of Lines of Actions. We test their idea in respect to Chess by testing for every move that gives check, whether the resulting position is mate or not. If a mate was found, the corresponding move is played. Only if no mate was found a move is chosen randomly.

[4] obtainable from: https://sites.google.com/site/gaviotachessengine/Home

4.2.2 Further modifications

Additional to modifications that aim at increasing the simulation accuracy, we also test a modification to the selection strategy (Progressive Bias) and a game theoretical modification (Monte-Carlo Solver).

- **Progressive Bias:** If a node was only visited a few times, its winning probability gives only little information about the strength of the corresponding move. Therefore, we apply Progressive Bias by storing for every node the result of Stockfish's heuristic evaluation function of that node and by modifying the selection strategy according to Formula 4 in Section 2.2.3. We do not test Progressive Bias in combination with the Static Exchange Evaluator. The SEE rates all non-capturing moves equal if they do not allow captures on the target square and would therefore have less influence on the selection strategy. Additionally, the speed improvement of SEE over the heuristic function has little influence on the speed of iterations, because we calculate the score only once per iteration.

- **Monte-Carlo Solver:** Similar to the approach by Winands et al. (2008) we mark proven winning nodes by setting their total rewards to a high value and proven losing nodes by setting their total rewards to a high negative value. However, we do not use a constant value, $C_{provenWin}$, for all proven winning nodes and its negative $-C_{provenWin}$ for all proven losing nodes. Instead, we calculate the total reward of a proven node depending on its shortest proven distance to mate, x, and therefore use $C_{provenWin}(x)$ and $-C_{provenWin}(x)$, respectively, with

  $$C_{provenWin}(x) = C_{provenWin}(0) - x \tag{5}$$

  and $C_{provenWin}(0)$ being a fixed high value.
  This allows us to prove the shortest distance to mate by using a different update strategy if a terminal game state was detected. The additional update strategy is only called if the expanded node of the current iteration is already mate. The method is sketched in Algorithm 4.1:
  The total reward of a winning node is only overwritten if a quicker win was not already discovered (line 3-5). If the current node is losing, its parent node is winning (line 11-14). If the current node is winning things are a bit more complicated: If the parent node is not fully expanded (line 14-17) or one of the siblings is not proven losing (line 21-24) we switch to the normal update strategy. Otherwise, we identify the child of the parent node that has the longest distance to mate (line 27-29), increment its distance to mate (line 32-36) and use this value to update the (losing) parent node (line 37).
  Additionally, the regular update function was modified to prevent it from updating the total rewards of proven winning or losing nodes. The selection strategy was modified to play proven winning moves with very high probability and to play proven losing moves with very low probability.

Algorithm 4.1 montecarlotreenode.cpp

```
1: function updateProvenGamestate(sideToMove, value)
2:     visits ← visits + 1
3:     if not (sideToMoveWins(sideToMove, value) and abs(totalValue > value)) then
4:         totalValue ← value
5:     end if
6:
7:     if has no parent then
8:         return
9:     end if
10:
11:    if sideToMoveLoses(sideToMove, value) then
12:        parent.updateProvenGamestate(other(sideToMove), value)
13:        return
14:    end if
15:
16:    if parent is not fully expanded then
17:        continueWithNormalUpdate(value)
18:        return
19:    end if
20:
21:    farthestLoss ← totalValue
22:    for all sibling ∈ siblings do
23:        if not sideToMoveWins(sideToMove,sibling.totalValue) then
24:            continueWithNormalUpdate(value)
25:            return
26:        end if
27:        if abs(sibling.totalValue) < abs(farthestLoss) then
28:            farthestLoss ← sibling.totalValue
29:        end if
30:    end for
31:
32:    if farthestLoss > 0 then
33:        farthestLoss ← farthestLoss − 1
34:    else
35:        farthestLoss ← farthestLoss + 1
36:    end if
37:    parent.updateProvenGameState(other(sideToMove), farthestLoss)
38: end function
```

5 Evaluation

5.1 Base implementation

The performance of unmodified MCC was very bad. It was actually so bad that comparisons to Alpha-Beta based engines are uninspiring as unmodified MCC would be no match for - even mediocre - Minimax implementations. The tree growth of MCC is almost symmetrical, indicating that the engine has severe difficulties to distinguish between very good and very bad moves. An inability to distinguish between winning and losing moves was also observable in games of self-play: Pieces were often moved to squares where the opponent could capture them with huge advantage - but too often it did not make use of such opportunities.

5.1.1 Draws

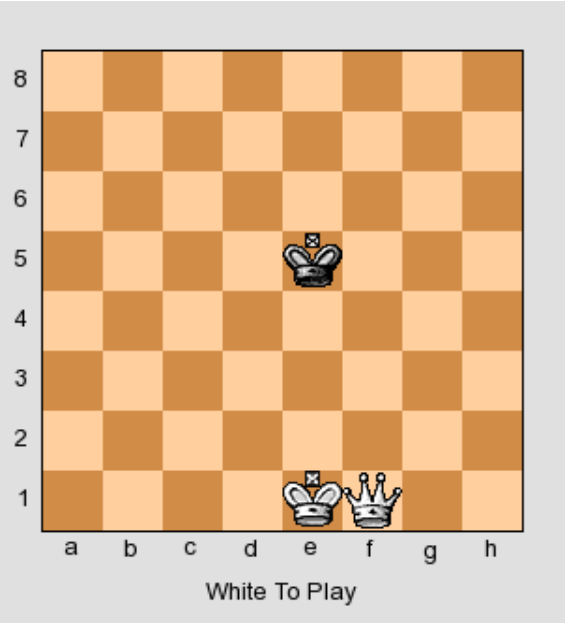

Figure 6: An endgame with king and queen against king

In order to better understand why the evaluation of good moves is often close to the evaluation of bad moves the influence of draws should be discussed. Draws are much more common in Chess than in the game of Go and in many clearly winning endgame positions the winning side needs to find the correct strategy in order to convert its advantage to a win.
Figure 6 shows a simple endgame where White has a king and a queen and Black only has a king. White is clearly winning as he is able to use both his pieces to first force the black king to the edge of the board and then mate it there. By following this plan, delivering mate is easy and can be demonstrated by chess beginners. What are the prospects of a random strategy to reach a mate in this position? In order to answer this question we performed a million simulations starting from the position in Figure 6. The overwhelming majority of these simulations ended in a draw. Only in 4988 simulations - about half a percent of all simulations - White was able to deliver a mate, leading to a winning probability of 50,2494%. The low winning probability is not surprising given the fact that *by chance* the black king must move to an edge while *by chance*

White must approach the black king with his own king and then *by chance* must play the mating queen move - all within fifty moves. The difficulties of the random simulation to deliver mate directly affect the evaluations of Monte-Carlo Tree Search which are almost always, except for rare cases, close to 50%.
One might argue, that these winning probabilities still correlate to the correct winning probabilities of the white player. However, it can be shown that this is not always the case.

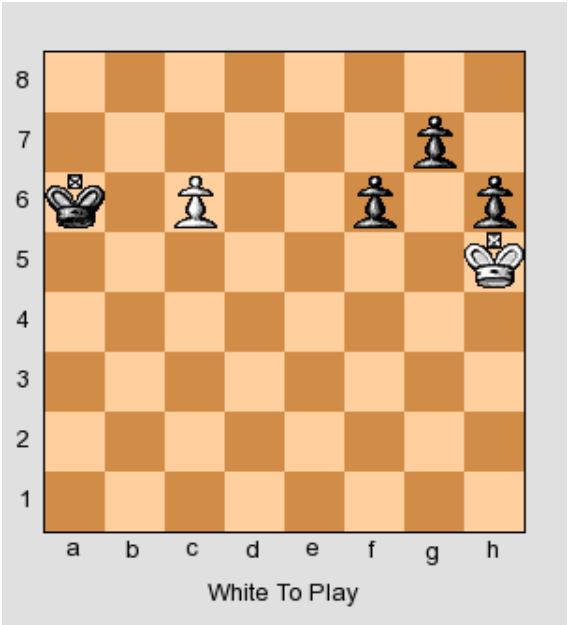

Figure 7: Famous study by Richard Réti. White is able to draw.

Figure 7 shows a famous endgame study by chess grandmaster Richard Réti. White is able to reach a drawn endgame by using the path g7-f6-e5 to simultaneously approach his own pawn and the promotion squares of the black pawns. The intrinsics of this endgame are not important for our considerations. However it should be stated that only by starting with 1.Kg6 White is able to defend a draw. A modified version of our base implementation, $MCC\_Combined_{3pc}$, which will be examined in Section 5.2, required in several tests always about two minutes to find 1.Kg6. Afterwards we repeated the tests with the following modification to the algorithm: Whenever a simulation ended in a draw, the result was discarded and the simulation was repeated. Only if after six simulations no win for either side was found, the algorithm continued as normal by updating the corresponding nodes with 0.5 - the value for a draw. It is needless to say that disregarding simulations in such a way is by no means efficient and reduces the speed of the iterations significantly. Interestingly, however, this modification was able to find 1.Kg6 within 10 seconds - much faster than $MCC\_Combined_{3pc}$. This observation can be explained under the assumption that the simulations on the positions arising after 1.Kg6 are less likely to *falsely* result in a draw than the simulations on the positions arising after a different first move (e.g. 1.Kg4). Repeating the simulations in the described manner has more effects on positions where the simulation is more likely to falsely result in a draw than on positions where the simulation is less likely to falsely result in a draw. Therefore the winning probability of 1.Kg6 was reduced less than the winning probabilities of other first moves. The described modification was only used to illustrate the effects of the affinity of the random simulation to end in a draw and is

not suited to increase the overall performance. However, as it is likely that random simulations in Chess more often falsely result in a draw than falsely result in a win or loss, it is tempting to weight draws less than wins. Xie & Liu (2009) weighted latter simulations higher by updating their results as if multiple simulations were performed with the same result.
The effects of using their update modification in order to weight differently depending on the simulation result have not yet been explored.

5.1.2 Accuracy

MCC's bad differentiations between good and bad moves indicate a low accuracy of the simulation strategy. We already showed on endgames that the affinity of the random strategy to reach drawn positions reduces its accuracy. However, the accuracy of a simulation strategy is affected by every move decision during the play-out and not only those made in the endgame. As MCC does not value material advantages high enough, it is interesting to see how material advantages influence the outcome of the simulation. We demonstrate that the features of a position that are usually good indicators for the evaluation of a position (e.g. material balance) are dominated in random simulations by other features that would be irrelevant for the outcome of a game between two reasonable players. UCT was modified for this demonstration so that it does not expand any nodes besides the root node. Therefore, all simulations begin after making one move in the root position. We used at least fifteen minutes per move giving rise to over three million iterations on our test system. Although, this is not enough to provide highly accurate estimates of the expected winning probabilities of the random simulation strategy and the winning probabilities should therefore be treated with some respect, they are accurate enough for our purpose.

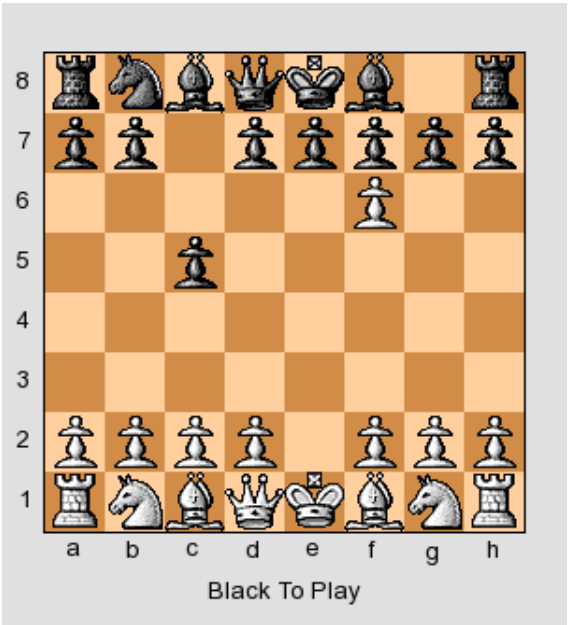

Figure 8: The starting position of our demonstration on the accuracy of the simulation strategy. White already has a winning advantage.

We start the demonstration at the position given in Figure 8 which could be reached from the starting position by 1.e4 Nf6 2.e5 c5 3.exf6. The second black move was a blunder that al-

lowed White to gain a material advantage which can be safely assumed as winning. Therefore, we would expect winning probabilities larger than 50%. The evaluations led to the following continuation: 3...gxf6(49.81%) 4.g4(50.02%) d5(49.78%) 5.f4(49.81%) Bxg5(49.33%) 6.Qxg4(49.96%). The numbers in the brackets show the estimated expected winning probability of the white player for the position reached after the corresponding move. We stop our demonstration after White won the second piece with 6.Qxg4. The final position is given in Figure 9.

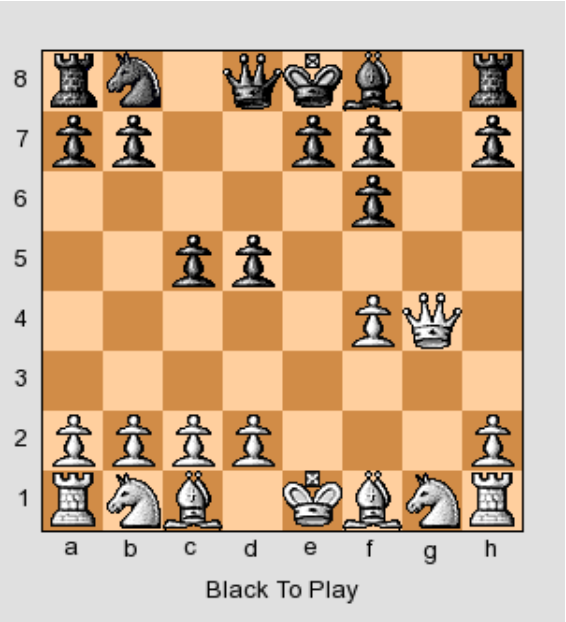

Figure 9: The final position of our demonstration of the accuracy of the simulation strategy. Although black lost a second piece, random simulations from this position still favor black slightly.

Although White should have a winning advantage in all positions that were encountered during the demonstration, the random simulation strategy favors Black in all positions beside the one reached after 4.g4. We spent over three hours in evaluating 6.Qxg4 and its winning probability should be very close to its expected value. White won a knight and a bishop for two pawns and there should be no doubt that its material advantage is decisive. Yet the random strategy favors Black slightly. The reasons for the bad evaluations are difficult to grasp due to the high game complexity. However, after observing several randomly played games from the final position we identified an important feature of the position: The positioning of the white queen. From g4 it can reach several squares where it can be captured. Especially, the option of moving to the square d7 is a major drawback of White's position because by playing Qd7+ black is forced to win the material. Such options would not affect the evaluation of a game that is played by reasonable players.
UCT would need to explore a position very deeply in order to eventually overcome the affects of such features of the position. As the inaccuracy of the random simulation strategy consistently steers the selection strategy towards bad branches, UCT is unable to build the desired asymmetrical tree structure.

5.2 Modifications

This section presents the results of the experiments on the performance of our modifications. We used round-robin tournaments of blitz chess with five minutes clock time per side for all our tests. The free tool BayesElo[5] by Rémi Coulom was used to calculate the Elo differences relative to the base implementation and their 0.95 confidence intervals. Elo ratings are used to rate the skill of chess players. Higher Elo ratings indicate better chess players. A player with an Elo rating that is by 200 points better than the rating of its opponent is supposed to score 75%.

5.2.1 Heavy Playouts

We tested our ϵ-greedy Heavy Playouts with the heuristic evaluation function ($MCC\_HP_{EVAL,\epsilon}$) and the Static Exchange Evaluator ($MCC\_HP_{SEE,\epsilon}$) independently as described in Section 4.2.1. For both variants we played 300 games in a round-robin tournament in order to determine the performance for different ϵ-values. Table 1 shows the performance of $MCC\_HP_{EVAL,\epsilon}$ for different values of ϵ. The corresponding likelihoods of superiority are given in Table 2.

Rank	Program	Elo	games	score
1	$MCC\_HP_{EVAL,0.6}$	153±62	100	62.5
2	$MCC\_HP_{EVAL,0.4}$	151±60	100	62.5
3	$MCC\_HP_{EVAL,0.2}$	110±62	100	55
4	$MCC\_HP_{EVAL,0.8}$	26±60	100	42.5
5	$MCC\_HP_{EVAL,0}$	5±61	100	39
6	MCC	0±61	100	38.5

Table 1: Performance of different ϵ-values for $MCC\_HP_{EVAL,\epsilon}$

ϵ	0.6	0.4	0.2	0.8	0	1
0.6		51	85	99	99	99
0.4	48		85	99	99	99
0.2	14	14		98	99	99
0.8	0	0	1		70	74
0	0	0	0	29		55
1	0	0	0	25	44	

Table 2: Likelihoods of superiority for different ϵ-values for $MCC\_HP_{EVAL,\epsilon}$

By using Heavy Playouts with the heuristic evaluation function the performance of MCC could be increased. The best performance could be reached by using the random strategy and the greedy strategy with about equal probability. In this case the Elo rating of the Heavy Playouts modification is by about 150 points better than the base implementation. If the probability of playing according to the greedy strategy is too low, the modification does not make enough use of the heuristic evaluation function. If the probability of the playing according to the greedy strategy is too high, the modification becomes too deterministic. For $\epsilon = 0.4$ and $\epsilon = 0.6$ the

[5] http://remi.coulom.free.fr/Bayesian-Elo/

modification is with a probability of 99% superior to the extreme cases of $\epsilon = 0$ or $\epsilon = 1$. The performance of $MCC\_HP_{SEE,\epsilon}$ for different values of ϵ is shown in Table 3. The corresponding table of the likelihoods of superiority is given in Table 4.

Rank	Program	Elo	games	score
1	$MCC\_HP_{SEE,0.4}$	332±63	100	61.5
2	$MCC\_HP_{SEE,0.6}$	324±61	100	60.5
3	$MCC\_HP_{SEE,0.2}$	326±64	100	60
4	$MCC\_HP_{SEE,0}$	270±63	100	52
5	$MCC\_HP_{SEE,0.8}$	260±61	100	51
6	MCC	0±77	100	15

Table 3: Performance of different ϵ-values for $MCC\_HP_{SEE,\epsilon}$

ϵ	0.4	0.2	0.6	0	0.8	1
0.4		56	57	93	96	99
0.2	43		51	90	94	99
0.6	42	48		90	94	99
0	6	9	9		60	99
0.8	3	5	5	39		99
1	0	0	0	0	0	

Table 4: Likelihoods of superiority for different ϵ-values for $MCC\_HP_{SEE,\epsilon}$

The performance gains by using the Static Exchange Evaluator turn out to be larger than the gains reached by using the heuristic evaluation function. All tested values for ϵ could significantly improve upon the performance of the base implementation - even the deterministic variation.

5.2.2 Endgame Tablebase

We created two variants of the Endgame Tablebase modification: MCC_GTB_{3pc} uses a 3-piece Endgame Tablebase and MCC_GTB_{4pc} uses a 4-piece Endgame Tablebase. As we are not interested in the performance gained by the direct usage of the tablebase, both variants completely ignore the tablebase if the position corresponding to the root node has less than eight pieces. Therefore both variants are not likely to make use of the tablebase at the very beginning of a simulation. A total number of 150 games was played in a round-robin tournament between MCC_GTB_{3pc}, MCC_GTB_{4pc} and the base implementation MCC. The resulting Elo estimates are shown in Table 5. The likelihoods of superiority are given in Table 6. Although, MCC_GTB_{4pc} is with a probability of about 95% superior to the base implementation the maximum likelihood Elo difference is small.

Rank	Program	Elo	games	score
1	MCC_GTB_{4pc}	56±55	100	58
2	MCC_GTB_{3pc}	13±55	100	47.5
3	MCC	0±55	100	44.5

Table 5: Performance of the Endgame Tablebase modifications compared to the base implementation

Program	MCC_GTB_{4pc}	MCC_GTB_{3pc}	MCC
MCC_GTB_{4pc}		91	95
MCC_GTB_{3pc}	8		65
MCC	4	34	

Table 6: Likelihoods of superiority for the Endgame Tablebase modifications and the base implementation

5.3 Comparison of the different modifications

We compared the different modifications in our main experiment. We did not test the Endgame Tablebase modification and the Monte-Carlo Solver modification independently because they do not provide significant performance gains. However we did test a combination of all our modifications, $MCC\_Combined$ which uses the 4-piece Endgame Tablebase (and makes full use of it), our Monte-Carlo Solver implementation, Heavy Playouts with a Static Exchange Evaluator and $\epsilon = 0.4$, Decisive Moves and Progressive Bias. By combining Monte-Carlo Solver with a 4-piece Endgame Tablebase support $MCC\_Combined$ is able to play 4-piece endgames perfectly. The other participants in the main experiment are MCC, $MCC\_DecisiveMoves$, $MCC\_HP_{SEE,0.4}$ and $MCC\_progressiveBias$. $MCC\_Combined$ and $MCC\_progressiveBias$ use $k_{bias} = 0.001$ to limit the influence of the heuristic function on the selection strategy. Prior tests have shown that this value is low enough to ensure that all children are selected several times but still high enough to provide measurable performance gains.
The results of the tournament is given in Table 7. $MCC\_Combined$ won all games, except for one that it lost to $MCC\_HP_{SEE,0.4}$. The best performance of a single modification was shown by $MCC\_DecisiveMoves$ which even performed better than $MCC\_HP_{SEE,0.4}$. $MCC\_progressiveBias$ did also improve upon the unmodified UCT player MCC and was ranked fourth.

Rank	Program	1	2	3	4	5	score
1	$MCC\_Combined$	-	25.0-0.0	24.0-0.1	25.0-0.0	25.0-0.0	99/100
2	$MCC\_DecisiveMoves$	0.0-25.0	-	16.5-8.5	21.5-3.5	21.0-4.0	59/100
3	$MCC\_HP_{SEE,0.4}$	1.0-24.0	8.5-16.5	-	16.5-8.5	20.5-4.5	46.5/100
4	$MCC\_progressiveBias$	0.0-25.0	3.5-21.5	8.5-16.5	-	16.5-8.5	28.5/100
5	MCC	0.0-25.0	4.0-21.0	4.5-20.5	8.5-16.5	-	17.0/100

Table 7: final table of the main experiment.

The corresponding Elo estimates relative to the base implementation are shown in Table 8. By combing all our modifications we were able to increase the performance by approximately 864 Elo. $MCC\_Combined$ encountered only five 4-piece endgames in all its matches and has comparable iteration speed to the unmodified version. Therefore we conclude that its increased performance was primarily achieved by using a more accurate simulation strategy. However, although, we were able to increase the performance of MCC significantly, our final version, $MCC\_Combined$, still does not stand a chance against Minimax based competitors. The likelihoods of superiority are shown in Table 9.

Rank	Program	Elo	games	score
1	$MCC\_Combined$	864±186	100	99
2	$MCC\_DecisiveMoves$	336±77	100	59
3	$MCC\_HP_{SEE,0.4}$	229±73	100	46.5
4	$MCC\_progressiveBias$	85±72	100	28.5
5	MCC	0±75	100	17

Table 8: Elo estimates according to the results of the main experiment.

Rank	Program	1	2	3	4	5
1	$MCC\_Combined$		99	99	99	100
2	$MCC\_DecisiveMoves$	0		98	99	99
3	$MCC\_HP_{SEE,0.4}$	0	1		99	99
4	$MCC\_progressiveBias$	0	0	0		97
5	MCC	0	0	0	2	

Table 9: Likelihoods of superiority for the main experiment.

6 Conclusion

Although Monte-Carlo Tree Search - and especially UCT - has great success in the game of Go, the algorithm is commonly considered to be unpromising for the game of Chess. This thesis was written in order to challenge this assumption. For this purpose, the chess program MCC was created and used to actually test the performance of the UCT algorithm in its simplest form with respect to Chess. For the same purpose, several possible enhancements to the algorithm were examined and tested.

Early assessments revealed that the main benefit of MCTS, namely the best-first search order, involves with its main drawback: The inability to identify search traps quickly. These assessments also indicate that without a simulation strategy that is accurate enough to lead to an asymmetrical tree growth the problem of identifying search traps is not encountered.

The tests with unmodified MCC show that a random simulation strategy in Chess is not sufficiently accurate to make use of the best-first search or to examine the problem of search traps. In order to better grasp the reasons for the bad accuracy of random playouts, it was demonstrated that the outcome of the playout mainly depends on positional features that a irrelevant for the outcome of a game that is played between reasonable players. It was also demonstrated, that the random playout has difficulties to deliver mate and therefore often falsely ends in a draw which does actually lead to worse accuracy. It was therefore proposed to reduce the weight of simulations that resulted in a draw.

With the aim of increasing the simulation strategy, ϵ-greedy Heavy Playouts and Decisive Moves were tested and a modification that uses Endgame Tablebases was proposed and tested. Additionally, Progressive Bias was tested and a modification of Monte-Carlo Solver that is able to prove the shortest mating distance was proposed and successfully used.

By combining all these modification the performance of MCC was increased by approximately 864 Elo points. The better part of these gains was reached by the Decisive Moves modification and by an ϵ-greedy Heavy Playout that used a Static Exchange Evaluator instead of the heuristic evaluation function. Although it was demonstrated that Endgame Tablebases can indeed increase the overall accuracy of the simulation strategy, the improvements by this modification were slim. The significant increase of playing strength that was reached by combining all improvements did still not suffice to compete with Alpha-Beta based chess programs or to examine the influences of search traps.

Nevertheless, this does not imply that combining MCTS and Chess is a dead end. Several ways to improve upon MCC were addressed, e.g. transposition tables or faster move generation. Additionally the improvements to the simulation strategy that were tested on MCC were of a relatively simple nature. More sophisticated ways to increase the simulation accuracy would most likely lead to further significant improvements. Once a sufficiently accurate simulation strategy is found, the problem of search traps can be tackled. The conclusion that the problem of identifying search traps quickly is inherent in the best-first search of MCTS does not imply that it can not be coped with. Alpha-Beta search is unable to provide good results at anytime - yet by combining it with Iterative Deepening search it is currently the most successful tree search algorithm with respect to Chess. Might a similar approach be effective for Monte-Carlo Tree Search, too?

7 Appendix

Algorithm 7.1 uctSearch.cpp

```
function UCT(Position rootPos, Time remainingTime)
    StopRequest ← false
    startTime ← currentTime
    thinkingTime ← remainingTime/timeRate
    root ← new MonteCarloTreeNode()
    while StopRequest = false do
        selected ← root.select(rootPos)
        expanded ← selected.expand(rootPos)
        result ← expanded.simulate(rootPos)
        expanded.update(result)
        iterations ← iterations + 1
        if iterations%1000 = 0 then
            StopRequest ← poll_for_stop(startTime, thinkingTime)
        end if
    end while
    return root.most_visited_child()
end function
```

Algorithm 7.2 montecarlotreenode.cpp

```
function SELECT(Position rootPos)
    currentPos ← getCurrentPosition(rootPos)
    blackToMove ← currentPos.isBlackToMove()
    cur ← this
    chosen ← this
    bestVal ← −1
    while cur.hasChildren() do
        if cur.isNotFullyExpanded then
            return cur
        end if
        for all child ∈ cur.children do
            winningRate ← (child.totalValue/child.visits)
            if blackToMove then
                winningRate ← 1 − winningRate
            end if
            uctVal ← winningRate + sqrt(2 ∗ log(cur.visits)/child.visits)
            if uctVal ≥ bestVal then
                bestVal ← uctVal
                chosen ← child
            end if
        end for
        currentPos.make_move(chosen.lastMove)
        blackToMove ← not blackToMove
        cur ← chosen
    end while
    return cur
end function
```

Algorithm 7.3 montecarlotreenode.cpp

```
function EXPAND(Position rootPos)
    currentPos ← getCurrentPosition(rootPos)
    if currentPos.isMateOrDraw() then
        return this
    end if
    availableMoves ← generate_moves(currentPos)
    if childen.size() = availableMoves.size() then
        return this
    end if
    return createAndAddChild(availableMoves[children.size()])
end function
```

Algorithm 7.4 montecarlotreenode.cpp

```
function SIMULATE(Position rootPos)
    currentPos ← getCurrentPosition(rootPos)
    while currentPos.isMateOrDraw() = false do
        availableMoves ← generate_moves(currentPos)
        index ← getRandomNumber() mod availableMoves.size()
        currentPos.make_move(availableMoves[index])
    end while
    if currentPos.whiteWon() then
        return 1
    end if
    if currentPos.blackWon() then
        return 0
    end if
    return 0.5
end function
```

Algorithm 7.5 montecarlotreenode.cpp

```
function UPDATE(double value)
    totalValue ← totalValue + value
    visits ← visits + 1
    if parent then
        parent.update(value)
    end if
end function
```

References

Adel'son-Vel'skii, G., Arlazarov, V., Bitman, A., Zhivotovskii, A. & Uskov, A. (1970), 'Programming a computer to play chess', *Russian Mathematical Surveys* **25**, 221.

Auer, P., Cesa-Bianchi, N. & Fischer, P. (2002), 'Finite-time Analysis of the Multiarmed Bandit Problem', *Machine Learning* **47**(2-3), 235–256.

Auer, P., Cesa-Bianchi, N., Freund, Y. & Schapire, R. E. (1995), Gambling in a Rigged Casino: The Adversarial Multi-Arm Bandit Problem, *in* 'FOCS', IEEE Computer Society, pp. 322–331.

Bellman, R. (1965), 'On the application of dynamic programing to the determination of optimal play in chess and checkers', *Proceedings of the National Academy of Sciences of the United States of America* **53**(2), 244.

Breuker, D., Uiterwijk, J. & Van Den Herik, H. (1977), Information in transposition tables, *in* H. J. van den Herik & J. Uiterwijk, eds, 'Advances in Computer Chess 8. Universiteit Maastricht, Maastrich.', pp. 199–211.

Chaslot, G., Saito, J., Bouzy, B., Uiterwijk, J. & Van Den Herik, H. (2006), Monte-Carlo strategies for computer Go, *in* 'Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium', pp. 83–91.

Chaslot, G., Winands, M., Herik, H., Uiterwijk, J. & Bouzy, B. (2008), 'Progressive strategies for Monte-Carlo Tree Search', *New Mathematics and Natural Computation* **4**(3), 343.

Coulom, R. (2006), Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, *in* H. J. van den Herik, P. Ciancarini & H. H. L. M. Donkers, eds, 'Computers and Games', Vol. 4630 of *Lecture Notes in Computer Science*, Springer, pp. 72–83.

Coulom, R. (2007), 'Computing "Elo Ratings" of Move Patterns in the Game of Go', *ICGA Journal* **30**(4), 198–208.

Drake, P. & Uurtamo, S. (2007), Move ordering vs heavy playouts: Where should heuristics be applied in monte carlo go, *in* 'Proceedings of the 3rd North American Game-On Conference'.

Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C. & Teytaud, O. (2012), 'The grand challenge of computer go: Monte carlo tree search and extensions', *Commun. ACM* **55**(3), 106–113.

Gelly, S. & Silver, D. (2007), Combining online and offline knowledge in uct, *in* Z. Ghahramani, ed., 'ICML', Vol. 227 of *ACM International Conference Proceeding Series*, ACM, pp. 273–280.

Gelly, S. & Silver, D. (2011), 'Monte-carlo Tree Search and Rapid Action Value Estimation in computer Go', *Artif. Intell.* **175**(11), 1856–1875.

Greenblatt, R., Eastlake III, D. & Crocker, S. (1967), The greenblatt Chess Program, *in* 'Proceedings of the November 14-16, 1967, Fall Joint Computer Conference', ACM, pp. 801–810.

Knuth, D. E. & Moore, R. W. (1975), 'An analysis of alpha-beta pruning', *Artif. Intell.* **6**(4), 293–326.

Kocsis, L. & Szepesvári, C. (2006), Bandit Based Monte-Carlo Planning, *in* J. Fürnkranz, T. Scheffer & M. Spiliopoulou, eds, 'ECML', Vol. 4212 of *Lecture Notes in Computer Science*, Springer, pp. 282–293.

Korf, R. E. (1985), 'Depth-First Iterative-Deepening: An Optimal Admissible Tree Search', *Artif. Intell.* **27**(1), 97–109.

Maynard Smith, J. & Michie, D. (1961), 'Machines that play games', *New Scientist* **12**, 367–9.

Meteopolis, N. & Ulam, S. (1949), 'The Monte Carlo Method', *Journal of the American statistical association* **44**(247), 335–341.

Ramanujan, R., Sabharwal, A. & Selman, B. (2010), On adversarial search spaces and sampling-based planning, *in* R. I. Brafman, H. Geffner, J. Hoffmann & H. A. Kautz, eds, 'ICAPS', AAAI, pp. 242–245.

Russell, S. J. & Norvig, P. (2010), *Artificial Intelligence - A Modern Approach (3. internat. ed.)*, Pearson Education. ISBN: 978-0-13-207148-2.

Saffidine, A., Cazenave, T. & Méhat, J. (2011), 'Ucd: Upper confidence bound for rooted directed acyclic graphs', *Knowledge-Based Systems* .

Schaeffer, J. (2000), 'The games computers (and people) play', *Advances in Computers* **52**, 189–266.

Silver, D. & Tesauro, G. (2009), Monte-carlo simulation balancing, *in* A. P. Danyluk, L. Bottou & M. L. Littman, eds, 'ICML', Vol. 382 of *ACM International Conference Proceeding Series*, ACM, p. 119.

Sutton, R. S. (1995), Generalization in reinforcement learning: Successful examples using sparse coarse coding, *in* D. S. Touretzky, M. Mozer & M. E. Hasselmo, eds, 'NIPS', MIT Press, pp. 1038–1044.

Tesauro, G., Rajan, V. T. & Segal, R. (2010), Bayesian inference in monte-carlo tree search, *in* P. Grünwald & P. Spirtes, eds, 'UAI', AUAI Press, pp. 580–588.

Teytaud, F. & Teytaud, O. (2010), On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms, *in* G. N. Yannakakis & J. Togelius, eds, 'CIG', IEEE, pp. 359–364.

Wang, Y. & Gelly, S. (2007), Modifications of UCT and sequence-like simulations for Monte-Carlo Go, *in* 'CIG', IEEE, pp. 175–182.

Winands, M. H. M. & Björnsson, Y. (2009), Evaluation Function Based Monte-Carlo LOA, *in* H. J. van den Herik & P. Spronck, eds, 'ACG', Vol. 6048 of *Lecture Notes in Computer Science*, Springer, pp. 33–44.

Winands, M. H. M., Björnsson, Y. & Saito, J.-T. (2008), Monte-Carlo Tree Search Solver, *in* H. J. van den Herik, X. Xu, Z. Ma & M. H. M. Winands, eds, 'Computers and Games', Vol. 5131 of *Lecture Notes in Computer Science*, Springer, pp. 25–36.

Xie, F. & Liu, Z. (2009), Backpropagation modification in monte-carlo game tree search, *in* 'Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on', Vol. 2, IEEE, pp. 125–128.