Combining Monte-Carlo Tree Search with state-of-the-art search algorithm in chess

Eine Kombinierung von Monte-Carlo Tree Search und State-of-the-Art Suchalgorithmus im Schach

Bachelor-Thesis von Alymbek Sadybakasov aus Bischkek

Tag der Einreichung:

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

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Combining Monte-Carlo Tree Search with state-of-the-art search algorithm in chess
Eine Kombinierung von Monte-Carlo Tree Search und State-of-the-Art Suchalgorithmus im Schach

Vorgelegte Bachelor-Thesis von Alymbek Sadybakasov aus Bischkek

1. Gutachten: Prof. Johannes Fürnkranz
2. Gutachten:
3. 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 August 31, 2016

(A. Sadybakasov)

Abstract

Monte-Carlo Tree Search (MCTS), a best-first search technique, works badly for chess. The goal of this thesis was to employ informed searches in the rollout phase in order to increase the performance of MCTS. For this purpose, a search function of a state-of-the-art chess engine (Stockfish) has been combined with MCTS-Solver. Several enhancements have been made in order to improve the main approach as well. Some enhancements performed significantly better than the naive approach. The proposed approaches particularly solved the problem of MCTS techniques in finding search traps in the game of chess.

Contents

1 Introduction **5**

2 Foundations **6**
2.1 Monte-Carlo Tree Search . . . 6
2.1.1 Basic algorithm . . . 6
2.1.2 MCTS-Solver . . . 6
2.1.3 UCT . . . 8
2.1.4 Shallow traps in MCTS . . . 8
2.2 Enhancements to MCTS . . . 9
2.2.1 General enhancements . . . 9
2.2.2 Enhancements to selection and expansion phases . . . 9
2.2.3 Enhancements to rollout phase . . . 10
2.2.4 Enhancements to backpropagation phase . . . 10
2.3 Algorithmic developments for chess . . . 10
2.3.1 Zero-sum games . . . 10
2.3.2 Minimax . . . 11
2.3.3 Alpha-beta pruning . . . 11
2.3.4 Iterative deepening search . . . 13
2.3.5 Quiescence search . . . 14
2.3.6 Negamax . . . 15
2.3.7 Razoring . . . 15

3 How Stockfish Works **16**
3.1 Node types . . . 16
3.2 Static evaluation function . . . 16
3.3 Main search function . . . 17
3.4 Local functions . . . 18

4 Description of the Proposed Approaches **19**
4.1 MCTS with informed rollouts . . . 19
4.1.1 Naive approach . . . 19
4.1.2 MCTS with bounded informed rollouts . . . 20
4.1.3 MCTS with random selected informed rollouts . . . 20
4.1.4 Influence of the remaining pieces . . . 20
4.1.5 MCTS in the middle and end games . . . 21
4.1.6 Tablebases . . . 22
4.2 Informed traverses in the selection and expansion phases . . . 22

5 Evaluation **24**
5.1 Experimental setup . . . 24
5.2 The rollout phase . . . 24
5.2.1 Naive approach . . . 24
5.2.2 Bounded informed rollouts . . . 25

5.2.3 Random selected informed rollouts 25
5.2.4 Remaining pieces 26
5.2.5 Phases 26
5.2.6 Tablebases 27
5.3 The selection and expansion phases 28
5.4 Comparison of the best approaches 28
5.5 Exploring the computation time 29

6 Conclusion and Future Work **31**

7 Appendix **32**

8 References **33**

1 Introduction

Monte-Carlo Tree Search (MCTS) technique has provided good results in Go, which has a huge search space of the moves. The technique performs a lot of random simulations or rollouts of mini games in each move and backpropagates the results along the search tree. However, chess still remains to be too challenging for it [Are12]. The main problem of it lied in identifying the search traps – critical positions where one of the players can make a decisive move. In addition, MCTS could hardly deliver checkmate in the endgames which was resulting in a draw in most cases. This can be partly explained by the fact that MCTS would consider only random moves when performing the rollout phase and the critical moves would be overlooked. Unlike in Go and due to the tactical nature, it becomes, thus, very hard for MCTS to find the best moves in the game of chess.

[Bai15] proposed some methods to try to overcome the problems of MCTS in finding search traps. In this work hybrid algorithms were introduced, employing more informed searches in each phase of MCTS rather than just exploring the random moves. The algorithms were tested in the games of Othello, Catch the Lion and 6×6 Breakthrough and provided significant performance gains over the MCTS baseline. A logical question comes up: will the same or similar method achieve equivalent results in chess?

This goal of this thesis is either to approve or to refute the above question. In contrast to the mentioned work and in order to perform more accurate informed searches, a search function of the state-of-the-art chess engine Stockfish will be employed in different phases. Although this thesis focuses primarily on the rollout phase, informed traverses in the selection and expansion phases – the phases that decide which nodes will be chosen to be expanded – will be tested as well. An additional parameter – the number of iterations in a fixed computation time – will be explored as well. In addition, all modifications will be compared to each other and the best ones will play a tournament with each other and with Stockfish as an additional competitor.

The following section provides the foundations of relevant search techniques and their improvements. A Monte Carlo Tree Search technique is covered and some possible enhancements to it are presented. The remainder of the section describes the algorithmic developments for chess, namely Minimax search and its enhancements.

Stockfish, a state-of-the-art chess engine, is described in Section 3, including design and schema of its static evaluation function, main search function and additional local functions.

Section 4 describes the proposed approaches to improve MCTS – informed rollouts and informed traverses in the selection and expansion phases. Several improvements to informed rollouts have been introduced as well.

Section 5 provides the evaluation results of proposed approaches.

Finally, Section 6 closes the thesis with a conclusion and discussion of possible directions for future work.

2 Foundations

2.1 Monte-Carlo Tree Search

2.1.1 Basic algorithm

Monte-Carlo Tree Search ([Cou07], [KS06], [CSB+06])(Figure 1) is a best-first search technique based on Monte-Carlo technique [MU49]. It uses stochastic simulations and constructs a game search tree for each move decision. The idea consists in playing simulations or rollouts of games by selecting pseudo random moves for each side until the simulated game reaches the terminal state.
The algorithm iteratively repeats following steps until some stopping criterion is met:

1. **Selection** – a tree policy or strategy is used to find the next most promising child node of the current game tree, in which the root node implies the current position. Only legal moves are used to find the child nodes. Child nodes that have been already visited (expanded) are not considered in this step anymore.

2. **Expansion** – after the game reaches the first position that is not presented in the tree yet, the state of this position will be added as a new node. As mentioned, there is only one expansion of the node for each simulated game in the tree.

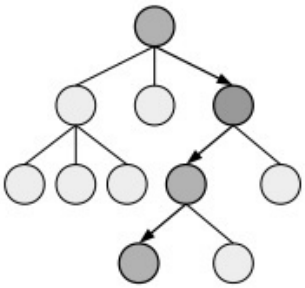
3. **Rollout** – starting from the expanded child node, a full game is played until the termination state has been reached. While domain-dependent improved rollout strategies are more extensive, they often need more resources than domain-independent strategies. This may be very critical in the domain of chess.

4. **Backpropagation** – reaching the terminal state means that the first player either has won, lost or the simulated game ended up in a draw. Wins are represented by 1, losses by 0 and draws by 0.5 respectively. MCTS backpropagates those values by updating each tree node on the path of the current game. The more times one node has been visited, the more score it gets.

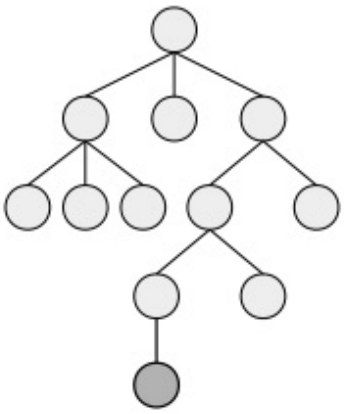
At the end, the algorithm executes the game action corresponding to the child node which was explored the most, namely the one with the greatest visit counts.
Pure MCTS does not evaluate the intermediate states during the rollout phase, while a Minimax search requires doing that for each single state. MCTS evaluates only the terminal states at the end of simulations in order to produce the backpropagation values. This works well in the domains where no long action sequences exist (e.g. Tic-Tac-Toe) but bad in chess. Still, MCTS can be applied to any game of finite length and backpropagates the outcome of each game immediately. This guarantees that all values are always ready to be returned and that the algorithm can be interrupted at any moment in time. With enough simulated games it will, thus, return acceptable results.

2.1.2 MCTS-Solver

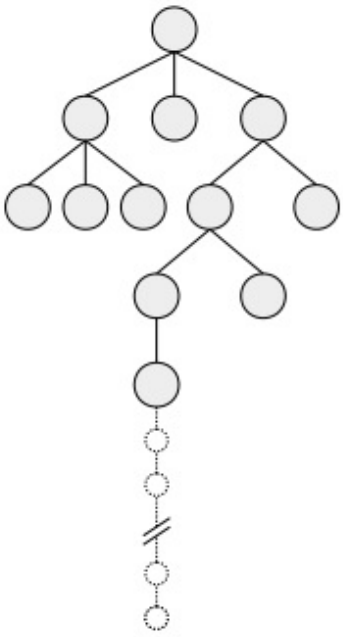
MCTS-Solver [WBS08] is an extension to MCTS with the basic idea marking the child nodes as proven losses and proven wins. If there is a node with a proven win for the first player, it will be



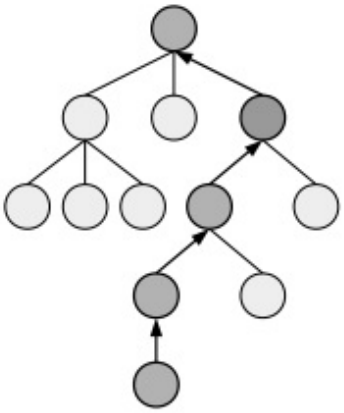
(a) The selection phase



(b) The expansion phase



(c) The rollout phase



(d) The backpropagation phase

Figure 1: Four phases of MCTS [Bai15]

marked as a proven win. This will lead to marking the previous node with a proven loss for the second player, which again leads to marking the previous node with a proven win for the first player, and so on. Generally, if all moves from the given state have been marked as proven losses for the first player, the previous state will be marked as a proven win for the second player.
This strategy has led to the significant gains in performance in e.g. Lines of Action [WBS08], Hex [AHH10], Havannah [Lor11], Shogi [STG10], Tron [DTW12], Focus [NW12], and Breakthrough [LH14].
The main benefit of MCTS-Solver compared to MCTS without this extension is that it does not re-evaluate already proven nodes, which leads to the time performance gains of simulations. It does not, though, avoid the weakness of MCTS in propagating all the way up through the tree.

2.1.3 UCT

As we have seen, a basic MCTS algorithm just gathers statistics for all children nodes of the root node. In order to get more accurate results, a deeper search is needed to be performed. Upper Confidence Bounds for Trees (UCT) [KS06] is the most popular algorithm in the MCTS family with the main idea to play sequences of nodes within some given time instead of considering each of them iteratively. The root nodes are represented as *independent bandits* and the child nodes as *independent arms*. UCT was proposed as an extension of UCB1 [ACBF02] – a classical deterministic selection policy which maximizes

$$UCT = X_j + \sqrt{\frac{2\ln n}{n_j}} \tag{1}$$

where n_j is the number of times a node j was played, X_j is the average winning probability for its child j and n is the overall number of plays so far. If two or more child nodes have the same UCT-value, the node will be selected randomly.
UCT has been widely used in the MCTS implementations due to its simplicity and proven convergence to the best move [KS06].

2.1.4 Shallow traps in MCTS

A *shallow* or *search* trap [RSS10] is a position, from which a winning/losing move for a player can be reached. In search spaces containing no or very few terminal positions (e.g. Tic-Tac-Toe), shallow traps should be rare and MCTS variants should make comparatively better decisions. In contrast, chess has a huge amount of terminal positions as well as 9 types of game pieces that move differently, which together with its tactical nature makes chess too challenging for MCTS techniques.
Search traps occur frequently in chess even at the high level. The *Légal Trap* is a basic example of such a trap. The trap is named after *Sire de Légal* who was the strongest chess player until 1755 when he lost a match to Philidor. The trap occurs at the opening, where White offers a queen sacrifice in order to try to checkmate Black in the next few moves. There are several variants of this trap, but we will focus on the main variant illustrated in Figure 2.
After **5. h3** the only move that can be played by Black is **5... Bxf3**. We assume that Black plays **5... Bh5?** instead, apparently trying to continue pinning the white knight. This move is, though, considered to be a blunder which leads to at least losing a pawn. White will response with **6. Nxe5!** and Black has to response with **6... Nxe5**, losing a pawn but being still in the game. If

Black takes the queen instead, White checkmates in two moves **6... Bxd1?? 7. Bxf7+ Ke7 8. Nd5#**.
Chess algorithms should ideally find the best move after 5. h3 in order to not fall into the trap. Modern chess engines typically do not have problems to find the best move. Since MCTS considers only random moves, it can take a lot of time until it will start to explore possible good moves and may be still not be able to find the only best move. MCTS can, thus, hardly overcome the problem. UCT could partly resolve this as explained in Section 2.1.3, but it would still not be able to compete with modern chess engines.

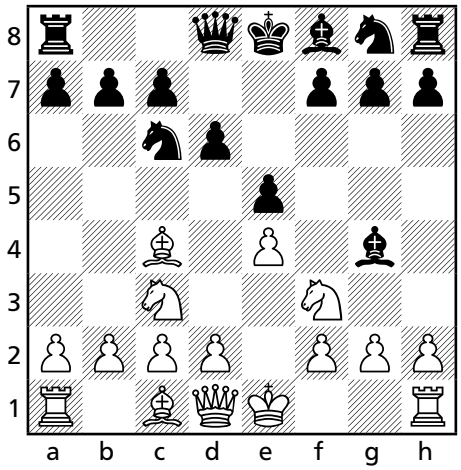


Figure 2: The Légal trap

2.2 Enhancements to MCTS

A number of enhancements to MCTS were proposed for different domains. This section gives an overview of the past researches in the area.

2.2.1 General enhancements

A logical improvement to the MCTS would be leaf, root and tree parallelizations [CWvdH08]. Another approach consists in regrouping nodes in the games with a huge branching factor [JC14], which should work good in games such as chess or Go. A deep learning approach to enhance the UCT algorithm with knowledge-based neural controllers by adjusting the probability distribution of UCT simulations was proposed by [XLW+10]. An interesting research work in investigating time management in Go was done by [BW12]. The best approach reached a significant improvement increasing the win rate by 39.1%.

2.2.2 Enhancements to selection and expansion phases

Informed minimax searches can be integrated in the selection and expansion phases [BW13], [Bai15]. This approach gained significant improvements over pure MCTS in games of Connect-4 and Breakthrough. The quality of the node selection also depends on the used policies. These policies can be divided into two types: *deterministic* and *stochastic*. Deterministic polices are

UCB1, UCB1-tuned, UCB-V, UCB-Minimal, OMC-Deterministic and MOSS, from which UCB1-tuned is considered to be the best in the game of Tron [PSPME12]. Stochastic policies are en-greedy, EXP3, Thompson Sampling, OMC-Stochastic and PBBM. Deterministic policies generally perform better than stochastic ones in Tron [PSPME12].

2.2.3 Enhancements to rollout phase

The promising enhancements can be theoretically achieved by using more intelligent moves in the rollout phase. [BW13], [Bai15] used fixed-depth minimax searches for choosing rollout moves. This approach should theoretically help to avoid certain types of blunders. However, time performance of such hybrid algorithms may be critical. [RT10] proposed an improvement by modifying the simulations depending on context. The improvement is based on a reward function learned on a "tiling of the space of simulations". It achieved good results in the game of Havannah with a winning percentage of 57% against the MCTS baseline. [SWU12] improved the rollout phase by using Last-Good-Reply and N-grams in the game of Havannah. A combined version achieved a win rate of 65.9% over other variants.

2.2.4 Enhancements to backpropagation phase

[BW13], [Bai15] employed shallow minimax searches actively searching for proven losses instead of hoping for MCTS-Solver to find them in future simulations. Just like in Section 2.2.2, this approach gained performance improvements over pure MCTS. A method to improve the performance of UCT by increasing the feedback value of the later simulations was proposed by [XL09]. This extension was tested in the game of Go and achieved a win rate of approximately 77% over UCT baseline.

2.3 Algorithmic developments for chess

Chess is classified as a zero-sum game and is characterized by deep variations for almost each position resulting to a very huge amount of possible positions. State-of-the-art chess engines typically do an exhaustive search of many such positions. Although their search process has already been optimized and the entire search space does not have to be examined, they still need to visit a huge amount of positions. A game search tree is built during this process with edges representing moves and nodes representing positions. Almost all strong chess engines are based on *minimax* and its enhancements as well as on some further optimizations.
The following sections provide an introduction to zero-sum games and describes the algorithmic developments for chess.

2.3.1 Zero-sum games

Zero-sum games is an area of game theory where two players are competing each other and trying to win. Both players can, thus, win, lose or draw the game and the results are opposed to each other. Chess engines typically have heuristic evaluation functions in order to evaluate the positions and return a $score$ as output. The resulting values are then used to determine whether a position is winning, losing or a draw for considered player. For example, $score = -10$ would mean that Black has a decisive advantage on the given position. If both players play only the best moves, then we are talking about the *principle variation*. A player's move is called *ply*, e.g. a chess engine with 6-ply depth will search only 6 moves ahead or, in other words, 3 moves for each side respectively.

2.3.2 Minimax

Minimax search [Neu28][Sha50] has been the most important search technique for a half of century. Almost all strong chess engines are based on this method along with further enhancements. Minimax search makes use of heuristic evaluation functions with the idea in considering the players either maximizing (Max player) or minimizing (Min player) the scores. This is also the way how a human thinks: "If I go there, then my opponent will have a winning move, so I will go rather differently". Figure 3 illustrates the process how a Min player minimizes its score. The pseudo code is given in Algorithm 1.
Instead of visiting all child nodes, it is possible to cut unpromising nodes, which is the core of the most important enhancement to Minimax search – alpha-beta pruning.

Algorithm 1 Minimax [RN09]

```
1: function MINIMAX-DECISION(position)
2:     return argmax_{a∈ACTIONS(positions)} MIN-VALUE(RESULT(position, a))
1: function MAX-VALUE(position)
2:     if (TERMINAL-NODE(position)) then return EVALUATE(position)
3:     for all (a in ACTIONS(positions) do
4:         v ← MAX(v, MIN-VALUE(RESULT(position, a)))
5:     return v
1: function MIN-VALUE(position)
2:     if (TERMINAL-NODE(position)) then return EVALUTE(position)
3:     for all (a in ACTIONS(positions) do
4:         v ← MIN(v, MAX-VALUE(RESULT(position, a)))
5:     return v
```

2.3.3 Alpha-beta pruning

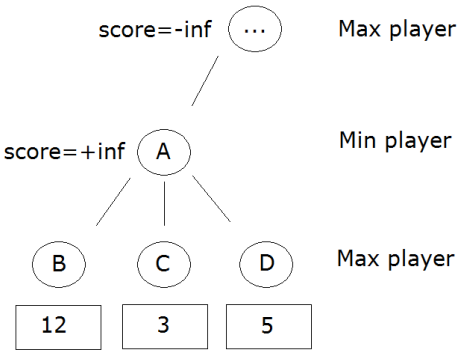
Alpha-beta pruning or α-β *pruning* [EH61] is an enhancement to Minimax search that makes use of information gained about part of the tree nodes to reject those branches which will not affect choosing the best moves. The method manages two values, namely α and β, and performs α- and β-cutoffs in the following manner:

- **α-cutoff** – occurs if Min player's score is less than or equal to α. Figure 4 illustrates the process.
- **β-cutoff** – occurs if Max player's score is greater than or equal to β.

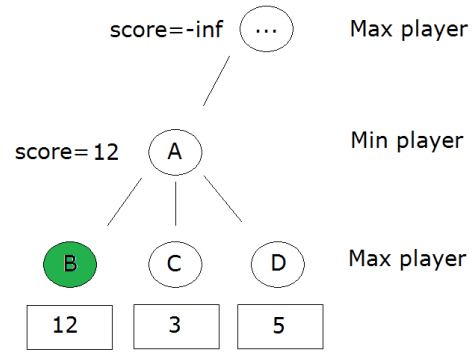
In case of perfect ordering, the α-β pruning method cuts the search tree in half, allowing almost doubling the search space compared to Minimax search when using the same computation time. The following theorem provides a theoretical base for this statement [EH61]:

Theorem 1 *(Levin): "Let n be the number of plies in a tree, and let b be the number of branches at every branch point. Then the number of terminal points on the tree is*

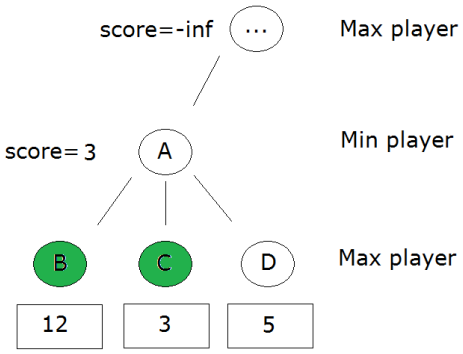
$$T = b^n$$



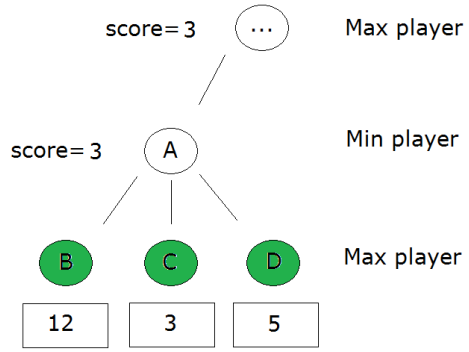
(a) Starting from the root node A, the algorithm expands the next child node B.



(b) Because B is a leave, a value 12 is backpropagated to the parent node, since $12 < +\infty$.



(c) The algorithm expands the next child node C and compares its value with the current score. Because $3 < 12$, the value is backpropagated to the parent node.



(d) The algorithm expands the next child node D and compares its value with the current score. Since $5 \not< 3$, the current score remains the same and will be then compared to the current of the higher parent node. Since $3 > +\infty$, the value is backpropagated to the higher parent node.

Figure 3: Minimax search: an example of score minimizing performed by Min player

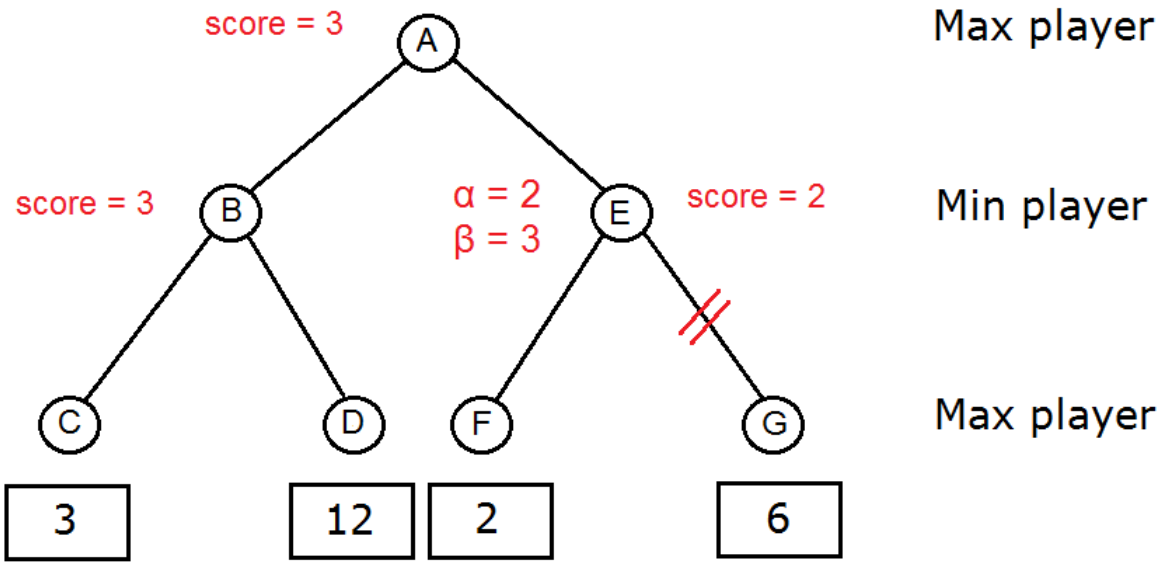


Figure 4: An example of an α-cutoff in which a node F has been recently visited. Since score < α, there is no more need in exploring the next nodes and the node G can be, thus, pruned.

However, if the best possible advantage is taken of the α-β heuristic then the number of terminal points that need be examined is

$$T = b^{(n+1)/2} + b^{(n-1)/2} - 1, \quad \textit{for odd } n$$
$$T = 2b^{(n/2)-1}, \quad \textit{for even } n"$$

2.3.4 Iterative deepening search

Iterative deepening search [Sco69](Algorithm 2) is a depth-limited search and has been used as the solution of difficulties of depth-first searches in managing the time. Starting with a 1-ply search, it gradually raises the depth window as soon as the previous search has been finished. The process continues until the allotted time has run out. Figure 5 shows the first four iterations of the iterative deepening search on a binary search tree.
Iterative deepening search has been widely used in chess engines due to the large search space and the uncertainty of the results. In some modifications in case of an unfinished search, the program always has the option to return the move selected in the last iteration of the search.

Algorithm 2 Iterative deepening search algorithm [RN09]

```
1: function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
2:     for (depth ← 0 to ∞) do
3:         if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
4:     return failure
```

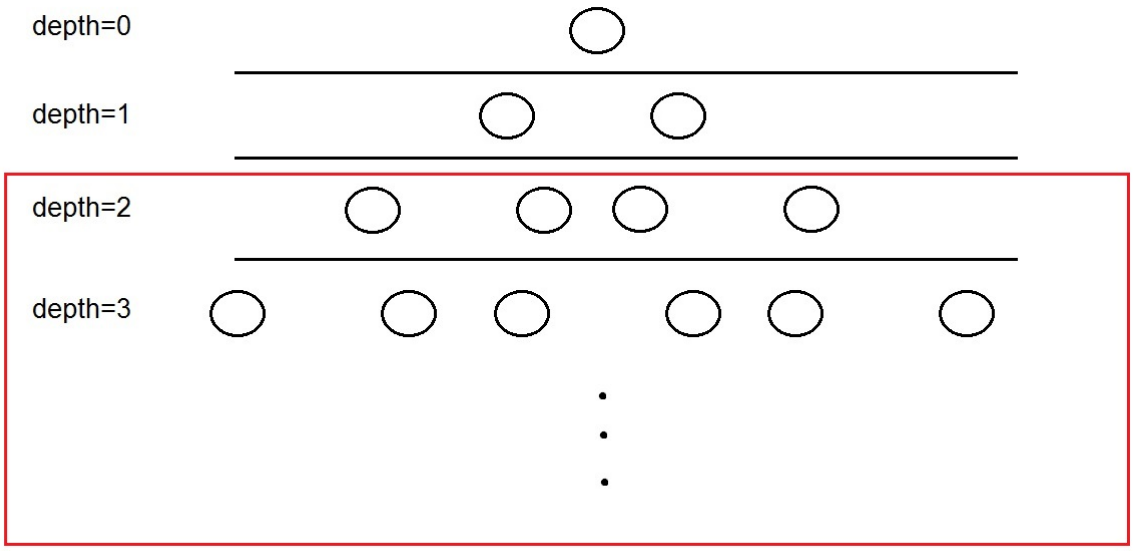


Figure 5: Iterative deepening search algorithm. Two iterations have already been performed. Red marked box denotes the depths to be explored.

2.3.5 Quiescence search

Modern chess engines perform a *quiescence search* [Har88] when the remaining depth is already near zero. This search considers only decisive moves such as captures, checks or checkmates. By investigating those moves and filtering them out, it assures that only "quiet" moves remain. This way, a *horizon effect* can be avoided. An example of horizon effect in chess is illustrated in Figure 6, with Black to move. We can see that the black bishop is trapped and the position can be evaluated as lost. If we suppose that a chess engine sees only 6 plies ahead, then it will try to prolong the bishop's life by pushing the black pawns and checking the white king. These moves are considered as good ones, while any other chess engine with a greater depth will consider the position as lost.

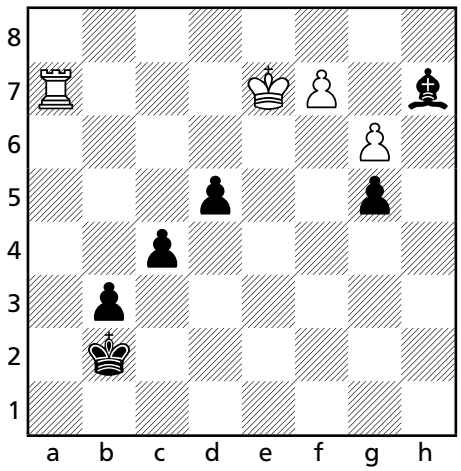


Figure 6: The horizon effect

2.3.6 Negamax

Negamax search [Smi95](Algorithm 3) is the variant of Minimax search and is usually used for simplicity, managing a single value for both players. To obtain the value for Min player, Negamax search just negates the general value, while Max player gets the same value without negation. There is, thus, no more need in computing both values for Max and Min players separately. The following mathematical relation describes the algorithm formally:

$$max(a,b) == -min(-a,-b) \tag{2}$$

Algorithm 3 Negamax [CCP]

```
1: function NEGAMAX(int depth)
2:     if (depth == 0) then return evaluate()
3:     int max ← −∞
4:     for all (moves) do
5:         score ← −negaMax(depth -1)
6:         if score > max then
7:             max ← score
8:     return max
```

2.3.7 Razoring

Razoring [BK88] is a technique for speeding up Minimax search. While α-β pruning guarantees to always find the best move, razoring does not, though, provide this functionality, it will, however, find an optimal move way faster than α-β pruning. In an n-ply tree razoring will prune plies 1 to n-1 and still find the optimal move, even though not the best one. By combining both techniques, a chess engine can achieve very good results in limited time. Razoring can also be extended to cut deeper trees by comparing backpropagated values from e.g. ply 4 of a four-ply program, with the static values at ply 1. This method is called *Deep Razoring* [Hei98].

3 How Stockfish Works

Stockfish [1] is currently the strongest open-source chess engine in the world [2], written in C++ and being developed by a community of chess developers.
Just like other modern chess engines, Stockfish does an exhaustive Minimax search with several further optimizations on it. Stockfish has a built-in static position evaluation function which can be used anytime in the game. The engine also manages transposition tables in order to evaluate the position which has been resulted from different move sequences only once instead of evaluating it many times. The transposition tables are implemented using a hash table with chess positions as keys. Stockfish uses a UCI Protocol [3] to communicate with GUIs and other chess engines.

3.1 Node types

Each node represents a chess position. The edges between the nodes are the corresponding moves. Stockfish uses the following types of nodes:

- **Fail-high nodes** are nodes in which a β-cutoff has been performed.
- **Fail-low nodes** are nodes in which no move's score has exceeded α.
- **PV nodes** are nodes that are lying on the principal variation path.

3.2 Static evaluation function

Stockfish uses an internal static evaluation function in order to find out how good a chess position is. This function takes into account many factors that can affect the evaluation, such as:

- **Raw material** – in chess, if one side has more pieces, this usually means a better position.
- **General piece placements** – each side should not just pay attention to the amount of pieces, but also try to control center squares, defend important friendly pieces or attack important enemy pieces.
- **Additional piece placements** – two bishops are usually better than two knights in the end game phase. There are a lot of such details that can affect the actual game flow.

These are not all factors, but enough to understand the functionality of static evaluation functions. They have been optimized over the years and now evaluate the chess positions almost perfectly. Those optimizations can be considered as one of the reasons that make Stockfish stronger than the other chess engines.
The static evaluation function does not assume the places where the pieces can be moved in the future. It is only considered to be a an additional help resource for the main search function.

[1] https://stockfishchess.org/
[2] http://www.computerchess.org.uk/ccrl/404/
[3] http://www.shredderchess.com/chess-info/features/uci-universal-chess-interface.html

3.3 Main search function

After the static evaluation function has evaluated the position, a main search function can examine all legal moves available and pick the best one. This move will then be made in the actual game. The main search function iteratively repeats the following steps:

1. *Node initializing*. A root node, from where the function will begin to search, is initialized.
2. *Mate distance pruning*. Even if Stockfish found a mate in the next move, the best score for it would be $value\_mated\_in$(state → ply) for white and $value\_mated\_in$(state → ply+1) for black respectively. But if α is already bigger or β is already less, because a shorter mate was found upward in the tree, there is no need to search further.
3. *Transposition table lookup*. As mentioned, transposition tables are used to avoid searching several lines that lead to the same position. At PV nodes Stockfish checks for exact scores, while at non-PV nodes it checks for a fail high/low node.
4. *Razoring*. Razoring prunes branches forward, if the static evaluation computed is less than or equal to α. In Stockfish it is 4*one_ply
5. *Quiescence search*. This search will be done when the remaining depth is less than one_ply.
6. *Static null move pruning*. This step reduces the search space by trying a *null* or *passing* move. It then checks if the score of the subtree search is still high enough to cause a β-cutoff.
7. *Null move search with verification search*. This step reduces dynamically null moves based on depth and on value.
8. *Internal iterative deepening search*. This step recursively calls the main search function with an increasing depth until there is no time or a maximum depth has been reached.
9. *Loop through moves*. After the previous step, the search function loops through all legal moves until no moves remain or a β-cutoff occurs.
10. *Decide the the new search depth*. After a candidate move was found, a singular extension search would be applied.
11. *Make the move*. A virtual move is made. If the move is neither capture nor promotion, then it will be added to the searched moves.
12. *Reduced depth search*. If the move made in the previous step fails high, Stockfish will make a re-search at full depth. This step will not be run if the initial depth is 1. If a reduced search returns a value very below β, we can (almost) safely prune the bad capture.
13. *Full depth search*. This step is only then applied if the move fails high.
14. *Undo move*. This step undoes the move made at the step 11.
15. *Check for a new best move*. The computed α and β values remain valid.
16. *Check for mate and stalemate*. Since all legal moves have been searched, only mate or stalemate are remaining. If one move was excluded, a fail-low score would be returned.

After the time limit is reached or no better moves can be found, the main search function will return a move with the best score found so far and it will be selected as the next move in the actual game to play.

3.4 Local functions

Stockfish has a number of local functions that can be tuned in order to get either performance or accuracy gains.

- **qsearch()** is the quiescence search function called by the main search function when the remaining depth is zero or, to be more precise, less than one_ply. It uses the transposition table lookups for move ordering and evaluates the position statically with α-β pruning. Since the situations where the remaining depth is zero often occur in the endgames, $qsearch()$ does not search useless checks to boost the performance.
- **check_is_dangerous()** tests if a checking move can be pruned in $qsearch()$ function. The best value is updated only when returning false, because in that case the move will be pruned. Considered moves are as follows:
  - Checks which leave at most one escape square for the opponent's king.
  - Checks by queen.
  - Double threats with checks.

  The best value will be updated only if check is not dangerous because $qsearch()$ will prune the move.
- **connected_moves()** checks whether two moves are connected in the sense that the first move somehow made the second move possible (for instance if the moving piece is the same in both moves). The first move is assumed to be the move that was made to reach the current position, while the second move is assumed to be a move from the current position.
- **extension()** is a function that decides whether a move should be searched with normal or extended depth. Certain classes of moves, such as checking moves, are searching with bigger depth than ordinary moves and are, in any case, marked as dangerous.

4 Description of the Proposed Approaches

The main goal of this thesis was to explore the results of combining the search function from a state-of-the-art chess engine with the MCTS-Solver in the domain of chess. The integration of MCTS-Solver into Stockfish was already done by [Are12] and was called MCC-Solver standing for Monte-Carlo Chess Solver. We used it as the baseline and compared it to all proposed hybrid algorithms. By using the search function we assumed the availability of heuristic evaluation functions. We, thus, extended [BW13] where no heuristic evaluation function had been used. In this section, we will describe two different approaches for employing heuristic knowledge within MCTS. The proposed approaches have not been tested in the game of chess yet. Although we have concentrated us on the rollout phase, informed searches in the selection and expansion phases will be considered as well in order to compare the different approaches. We also proposed several variations of employing informed rollouts in order to improve its performance.

4.1 MCTS with informed rollouts

4.1.1 Naive approach

It has been proven that UCT converges to the optimal policy [KS06]. However, more informed rollout strategies typically greatly improve performance [GWMT06]. For this reason, it seems natural to use an informed search function for choosing moves in the each rollout. By using the search function of Stockfish, we are able to find the best move from each given position. Search time for each rollout's move was limited according to the overall remaining time.
The proposed algorithm is based on MCC-Solver: in each rollout the function $simulate()$ checks whether the position is already a mate or a draw. It then computes the list of available moves provided by Stockfish. By calling the method $think(position, availableMoves)$, it starts to analyze the position using Stockfish's internal search function. On finish, it makes the move found by the search function. After the rollout is completed, the function returns the result: 1 if white wins, 0 if black wins and 0.5 in case of a draw. The pseudo code is given in Algorithm 4.
Our proposed strategy should improve the quality of play in the rollouts by avoiding certain types of blunders and search traps. It informs tree growth by providing more accurate rollout returns. We call this strategy *MCC-SR* standing for Monte-Carlo Chess with Stockfish rollouts.

Algorithm 4 Naive approach

```
1:  function SIMULATE(Position rootPos)
2:      currentPos ← getCurrentPosition(rootPos)
3:      while (currentPos.isMateOrDraw() = false) do
4:          availableMoves ← generate_moves(currentPos)
5:          index ← think(currentPos, availableMoves)
6:          currentPos.make_move(availableMoves[index])
7:      if currentPos.whiteWon() then
8:          return 1
9:      if currentPos.blackWon() then
10:         return 0
11:     return 0.5
```

4.1.2 MCTS with bounded informed rollouts

The main problem of employing informed search function in the rollout phase lies in its performance. In a blitz match against the MCC-Solver, MCC-SR would have bad chances due to the lack of simulated games. For this reason and to compare the pure performance of both strategies, we limited the number of iterations to k playouts, with k ∈ 500, 1000, 2000, 5000 for both MCTS-SR and MCC-Solver. To do this, we added a condition to the while-loop in the function UCT. The pseudo code is provided in Algorithm 5. We call this strategy *MCC-SR-k*, where *k* means the number of iterations.

Algorithm 5 Rollouts limited to 5000

```
1:  function UCT(Position rootPos, Time remainingTime)
2:      currentPos ← getCurrentPosition(rootPos)
3:      StopRequest ← false
4:      startTime ← currentTime
5:      thinkingTime ← remainingTime/timeRate
6:      root ← newMonteCarloTreeNode()
7:      while (StopRequest = false and iterations <= 5000) do
8:          selected ← root.select(rootPos)
9:          expanded ← selected.expand(rootPos)
10:         result ← expanded.simulate(rootPos)
11:         expanded.update(result)
12:         iterations ← iterations + 1
13:         StopRequest ← poll_for_stop(startTime, thinkingTime)
14:     return root.most_visited_child()
```

4.1.3 MCTS with random selected informed rollouts

Looking ahead, we can say that using Stockfish's search function in every rollout is very time expensive. Having for example only a few seconds for each move remaining, MCC-SR would play only a small number of simulations and, thus, have only a poor knowledge of the explored moves. Therefore, we changed our baseline algorithm MCC-SR by calling the search function only in several searches according to the given probability *p*. This would mean that having *p* = 20% our algorithm would call the search function only in 20% of the overall rollouts. In all other cases, the algorithm will choose the random move. We call this strategy *MCC-SR-PR-p* where *p* stands for the probability value. The pseudo code is provided in Algorithm 6.

4.1.4 Influence of the remaining pieces

In our preliminary tests we noticed that calling the search function consumes a different amount of time in different game positions. One possible factor might be the number of the remaining pieces on the board. In order to prove, we propose a strategy which calls the search function according to the given pieces on the board. It uses the Stockfish's internal function $get\_remaining\_pieces(position)$ to get the appropriate value. We, thus, check in each rollout whether there are a specific amount of pieces on the board remaining. We chose 3 different configurations: *MCC-SR-32-25-R* standing for calling the search function if there are more than 25

Algorithm 6 Informed rollouts with probability of 20%

```
1:  function SIMULATE(Position rootPos)
2:      currentPos ← getCurrentPosition(rootPos)
3:      while (currentPos.isMateOrDraw() = false) do
4:          availableMoves ← generate_moves(currentPos)
5:          prob ← random() mod 100
6:          if prob = 20 then
7:              index ← think(currentPos, availableMoves)
8:          else
9:              index ← getRandomNumber() mod availableMoves.size()
10:         currentPos.make_move(availableMoves[index])
11:     if currentPos.whiteWon() then
12:         return 1
13:     if currentPos.blackWon() then
14:         return 0
15:     return 0.5
```

pieces on the board, *MCC-SR-25-15-R* if there are 15 to 25 pieces on the board and *MCC-SR-15-R* if there are less than 15 pieces on the board remaining. The pseudo code of MCC-SR-32-25-R is given in Algorithm 7.

Algorithm 7 MCC-SR-32-25-R

```
1:  function SIMULATE(Position rootPos)
2:      currentPos ← getCurrentPosition(rootPos)
3:      while (currentPos.isMateOrDraw() = false) do
4:          availableMoves ← generate_moves(currentPos)
5:          remaining_pieces ← get_remaining_pieces(currentPos)
6:          if remaining_pieces > 25 then
7:              index ← think(currentPos, availableMoves)
8:          else
9:              index ← getRandomNumber() mod availableMoves.size()
10:         currentPos.make_move(availableMoves[index])
11:     if currentPos.whiteWon() then
12:         return 1
13:     if currentPos.blackWon() then
14:         return 0
15:     return 0.5
```

4.1.5 MCTS in the middle and end games

Our previous strategy contained the function that counts the number of pieces on the board. This can be not enough to separate between the different phases of the game. We, thus, propose the next strategy that calls the search function only in certain phases. Stockfish's embedded function $get\_current\_phase(position)$ considers only middle and end games. Openings and

middle games are, thus, considered as the same phase. We call our approaches *MCC-SR-MG* standing for calling a search function only in the middle games and *MCC-SR-EG* in the end games. The pseudo code is given in Algorithm 8.

Algorithm 8 Checking whether the current phase is a middle game

```
1:  function SIMULATE(Position rootPos)
2:      currentPos ← getCurrentPosition(rootPos)
3:      while (currentPos.isMateOrDraw() = false) do
4:          availableMoves ← generate_moves(currentPos)
5:          phase ← get_current_phase
6:          if phase = middle game then
7:              index ← think(currentPos, availableMoves)
8:          else
9:              index ← getRandomNumber() mod availableMoves.size()
10:         currentPos.make_move(availableMoves[index])
11:     if currentPos.whiteWon() then
12:         return 1
13:     if currentPos.blackWon() then
14:         return 0
15:     return 0.5
```

4.1.6 Tablebases

It was shown that using the tablebases in the rollouts could gain some performance [Are12]. We, thus, extend our naive approach by using the tablebases in the end phases of games. This variant accesses to the tablebase if the position corresponding to the root node has less than eight pieces. Therefore, this variant is not likely to make use of the tablebase at the very beginning of a simulation.

4.2 Informed traverses in the selection and expansion phases

Some approaches for embedding minimax searches in the selection and expansion phases of MCTS were proposed by [BW13], [Bai15]. One most promising approach consisted in starting a minimax search as soon as the state had reached a given number of visits. For 0 visits, this would include the expansion phase. We propose a slightly different approach: the search function will start searching as soon as the leaf node has been reached. The rollout phase will, thus, simulate the games starting from expanded positions that were resulted by the "good" selected moves. We call this approach *MCC-SSE* standing for Monte-Carlo Chess with Stockfish's search function in the selection and expansion phases. The pseudo code is given in Algorithm 9.

Algorithm 9 Implementation of informed traverses

```
1: function EXPAND(Position rootPos)
2:     currentPos ← getCurrentPosition(rootPos)
3:     if currentPos.isMateOrDraw() then
4:         return 1
5:     availableMoves ← generate_moves(currentPos)
6:     index ← think(currentPos, availableMoves)
7:     return createAndAddChild(availableMoves[index])
```

5 Evaluation

5.1 Experimental setup

All tournaments and matches were played using the tool LittleBlitzer which was developed by Kimien Software. It allowed us to play multiple games at the same time using multiple process threads. We used 4 threads for each our experiment which would mean 4 parallel games at one time. The results were stored in a Portable Game Notation (PGN) format. This format is being used to export and import chess games and is compatible with common UCI-supported chess engines and GUIs. The results were then analyzed using the tool BayesElo developed by Rémi Coulum [4]. It estimates relative differences in Elo ratings of one player relative to another. Elo ratings are used to estimate the player's strength and are initialized as 0. The tool also calculates an overall percentage of scores and draws based on the given PGN file. Each game in the tournaments and matches was a blitz game with 5 minutes time for each side and no increment available. All tournaments and matches were run on a machine with an Intel Core i5 and 4 GB RAM.

5.2 The rollout phase

5.2.1 Naive approach

We analyzed the pure MCC-SR by letting it evaluate the main position of Légal Trap (Figure 2). After 5. h3, MCTS-SR needed only 11 seconds to find the only best move 5...Bxf3. Although MCC-Solver took in account the only optimal move, in the end it, however, chose some other random move. This could be explained by the fact that MCTS-SR simulated the moves that were found by the search function of Stockfish. Since there were only a limited amount of those moves in this position, MCC-SR chose the best move very fast. On the other hand, MCC-Solver played only random moves in the rollouts. The game tree of MCTS-SR hereby grew in height, while that of MCC-Solver grew in width, which explains its weakness in identifying such traps. We could, thus, observe that our naive approach had overcome the problem of search traps, which had been the main challenge for MCTS in the game of chess.
We played 100 games in an additional face-to-face match between MCC-SR and the baseline MCC-Solver. Both variants did not have a fixed number of UCT iterations. The results are provided in Table 1.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-Solver	100	69%	47%	112±28
2	MCC-SR	100	32%	47%	0±28

Table 1: Match result between between MCC-Solver and MCC-SR

As we can see, MCC-SR performed worse than its opponent. The main problem lied in the fact that it could perform only a very small amount of rollouts compared to MCC-Solver. We tried to overcome this problem by performing limited informed rollouts in Sections 5.3, 5.4 and 5.5.

[4] http://www.remi-coulom.fr/Bayesian-Elo

5.2.2 Bounded informed rollouts

We tested MCC-SR with bounded informed rollouts in order to check the influence of the number of UCT iterations. 250 games were played in a round-robin tournament. The results are provided in Table 2. As we can see, there is only a small difference in the Elo ratings. We expected that MCC-SR-5000 would perform better, but it performed even worse. We assumed that the difference between the amount of simulations was too small to achieve any significant performance boost.
Furthermore, we played an additional tournament with MCC-SR and MCC-Solver and let them to perform only 5000 simulations. The results are provided in Table 3.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR-1000	100	55%	60%	42±42
2	MCC-SR-2000	100	53%	60%	36±41
3	MCC-SR	100	50%	60%	21±41
4	MCC-SR-5000	100	47%	64%	9±41
5	MCC-SR-500	100	45%	54%	0±42

Table 2: Tournament results to test the variants with limited rollouts

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-Solver-5000	100	64%	37%	84±28
2	MCC-SR-5000	100	37%	37%	0±28

Table 3: Results of an additional match between MCC-Solver-5000 and MCC-SR-5000

Unfortunately, the results were not so promising as we expected. We noticed that 20% of the games were finished due to timeouts of MCC-SR-5000. By playing a more extensive tournament with e.g. 60 minutes for each side we might hope to achieve more accurate results.

5.2.3 Random selected informed rollouts

Since pure MCC-SR did not perform well, we tested the effect of applying the stochasticity in employing the search function in MCTS. For that purpose, we played 250 games in a round-robin tournament and MCC-SR as an additional competitor. All proposed variants had only time restrictions and no iterations limit. The results are given in Table 4. We can see that no significant improvement can be achieved by this approach. All methods performed even worse than our baseline. The best approach with 80% probability reached only 52% of winning games staying behind MCC-SR with 65% of winning games. We, thus, did not do any further tournaments. We also can see that the results are decreasing together with the probability.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR	100	65%	49%	104±44
2	MCC-SR-PR-80	100	52%	40%	47±44
3	MCC-SR-PR-40	100	45%	46%	18±43
4	MCC-SR-PR-60	100	46%	38%	16±45
5	MCC-SR-PR-20	100	43%	33%	0±46

Table 4: Tournament results to test the effect of stochasticity

5.2.4 Remaining pieces

Although we did not expect any improvements, this approach gave the most significant performance gains to MCC-SR. In our experiment we played a round-robin tournament with 200 games. All methods didn't have any restrictions on the number of UCT iterations. Tournament results are provided in Table 5.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR-32-25-R	100	83%	2%	828±63
2	MCC-SR-15-R	100	68%	1%	704±62
3	MCC-SR-25-15-R	100	49%	0%	547±62
4	MCC-SR	100	1%	1%	0±151

Table 5: Tournament results to test the influence of the remaining pieces

The difference in Elo ratings between MCC-SR-32-25-R and MCC-SR was approximately 828 points. This performance was the best among all our previous approaches so far. In order to measure the difference between MCC-SR-32-25-R and MCC-Solver and to prove the performance gains, we played an additional match with 100 games between them. The results are provided in Table 6.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR-32-25-R	100	100%	0%	722±127
2	MCC-Solver	100	0%	0%	0±127

Table 6: Results of an additional match between MCC-SR-32-25-R and MCC-Solver

As expected, this approach beat MCC-Solver baseline with a 100% result and an estimated difference in Elo ratings of approximately 722 points was achieved.

5.2.5 Phases

We have already shown that the influence of the remaining pieces may be big. We can, thus, assume that the same will happen if we take into account the different game phases of chess. To prove this we played 150 games in a round-robin tournament with MCC-SR, MCC-SR-MG and MCC-SR-EG as its competitors. Both proposed variants did not have any limits of the number of UCT iterations. The results are provided in Table 7.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR-EG	100	77%	5%	670±55
2	MCC-SR-MG	100	73%	4%	644±55
3	MCC-SR	100	1%	1%	0±134

Table 7: Tournament results to test the influence of the different phases

As assumed, these approaches could get performance gains as well. Although the difference between Elo ratings from MCC-SR-EG and MCC-SR-32-25-R was about 158 points, this could still be considered as a good result. Since the difference between MCC-SR-EG and MCC-SR-MG was only around 26 points, we could say that using informed rollouts in the middle or end game phases is equally good and both of them could provide significant performance boosts to the naive approach. We could also observe that the amount of decisive games was very high. Especially MCC-SR had only one draw and no wins at all. Most of the decisive games were ended up by delivering the checkmate in the middle games. Furthermore, all draws were ended in the end games. Although the resulted game sample was not so representative, MCTS with informed rollouts could still hardly overcome the problem of draws in chess as mentioned by [Are12].
To compare the best approach with the baseline we played an additional match with 100 games and chose MCC-SR-EG since it performed slightly better than MCC-SR-MG. The results are given in Table 7. MCC-Solver managed to win only one game and made no draws against MCC-SR-32-25-R. We, thus, could confirm that the number of remaining pieces on the board had a great impact on employing informed rollouts.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR-EG	100	99%	0%	656±100
2	MCC-Solver	100	1%	0%	0±100

Table 8: Results of an additional match between MCC-SR-EG and MCC-Solver

5.2.6 Tablebases

We played 100 games in a match between MCC-SR-TB and MCC-Solver in order test whether using tablebases could improve our baseline approach. We used Gaviota Endgame Tablebase [5] to stop simulations as soon as the endgame tablebase could be retrieved. The results are given in Table 9.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR-TB	100	62%	50%	72±127
2	MCC-Solver	100	38%	50%	0±127

Table 9: Match result between MCC-SR-TB and MCC-Solver

50% of games were ended up in a draw and only 12% of games were decisive. The resulting difference in Elo ratings was only about 72 points which discouraged us to do further tests.

[5] https://sites.google.com/site/gaviotachessengine/Home/endgame-tablebases-1

5.3 The selection and expansion phases

We implemented this approach to step away from the rollout phase and to test informed searches in other phases. To see how this approach would perform, we played two matches each of 100 games. It played against both baselines MCC-Solver and MCC-SR. The results are provided in Tables 10 and 11.
Just like MCC-SR-32-25-R, MCC-SSE won all games against both MCC-Solver and MCC-SR in which almost all games a checkmate was delivered. We could, thus, conclude that MCC-SSE was our second best approach that employing informed searches in the selection and expansion phases might be an interesting direction for further experiments. The fact that we did not do any improvements to this method could tell us that there could be a high potential to achieve even better results. We might assume that this approach with potential improvements could even perform well against Stockfish. However, since this thesis focused on the rollout phase, we didn't do any further enhancements on the proposed method.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SSE	100	100%	0%	722±127
2	MCC-Solver	100	0%	0%	0±127

Table 10: Match result between MCC-SSE and MCC-Solver

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SEE	100	100%	0%	722±127
2	MCC-SR	100	0%	0%	0±127

Table 11: Match result between MCC-SSE and MCC-SR

5.4 Comparison of the best approaches

In this section we would like to find out how good each best method would perform against each other. For this purpose, we chose the best methods from each previous section and let them play in a 400 games round-robin tournament. The results are given in Table 12.

Rank	Engine name	Games	Score	Draws	Elo
1	MCC-SR-EG	100	87%	1%	858±95
2	MCC-SR-32-25-R	100	86%	0%	849±96
3	MCC-SSE	100	86%	1%	840±63
4	MCC-Solver	100	39%	22%	201±63
5	MCC-SR-1000	100	32%	25%	149±63
6	MCC-SR	100	29%	20%	172±64
7	MCC-SR-PR-80	100	26%	26%	98±64
8	MCC-SR-TB	100	17%	19%	0±67

Table 12: Tournament results between the best methods and baselines

As we can see, our three best approaches have performed roughly speaking equally well. MCC-SR-EG performed slightly better, while MCC-SR-32-25-R played only decisive games without any draws.
Interesting results could be achieved by letting the best approaches to play against. We, thus, conducted 3 additional matches each of 100 games. The main purpose of this test was to find out the potential of the best approaches to overcome a state-of-the-art chess engine. The results are provided in Tables 13–15.

Rank	Engine name	Games	Score	Draws	Elo
1	Stockfish	100	72%	0%	192±36
2	MCC-SR-32-25-R	100	28%	0%	0±36

Table 13: Match result between MCC-SR-32-25-R vs. Stockfish

Rank	Engine name	Games	Score	Draws	Elo
1	Stockfish	100	79%	0%	264±39
2	MCC-SR-EG	100	21%	0%	0±39

Table 14: Match result between MCC-SR-EG vs. Stockfish

Rank	Engine name	Games	Score	Draws	Elo
1	Stockfish	100	79%	0%	264±39
2	MCC-SSE	100	21%	0%	0±39

Table 15: Match result between MCC-SSE vs. Stockfish

Although the results looked promising at first sight, we should pay attention to the fact that Stockfish had winning positions in almost all games it lost. As we had started to investigate lost games, we found out that Stockfish had, surprisingly, difficulties in managing time control and, thus, lost these games on time. We found only very few games, where no positional or decisive advantage was gained by Stockfish. Figure 8 illustrates the ending position of one of such games. We can see that neither White nor Black has a decisive advantage. The game just stopped due to no response from Stockfish. We couldn't find out the underlying problem.
Another example of Stockfish's loses is game 12 from the same match which is illustrated in Figure 9. White had a decisive advantage having a material advantage and aiming at the black queen. Suddenly, something happened and White lost on time. We did not find any explanation, why Stockfish had such difficulties in managing time control. We could only try to reproduce the problem on a GUI with UCI support and to see if the timeout would happen again. Since this attempt was unsuccessful, we assume that it could be a bug in LittleBlitzer that caused this problem. We might assume that Stockfish would overcome the variants in the lost games. This would mean that our proposed approaches would be not ready yet to compete with Stockfish, although they performed significantly better than our baselines.

5.5 Exploring the computation time

We measured the number of performed UCT iterations for MCC-SR-PR-p, MCC-Solver and MCC-SR within the fixed time. We did this in order to understand how calls to Stockfish's search

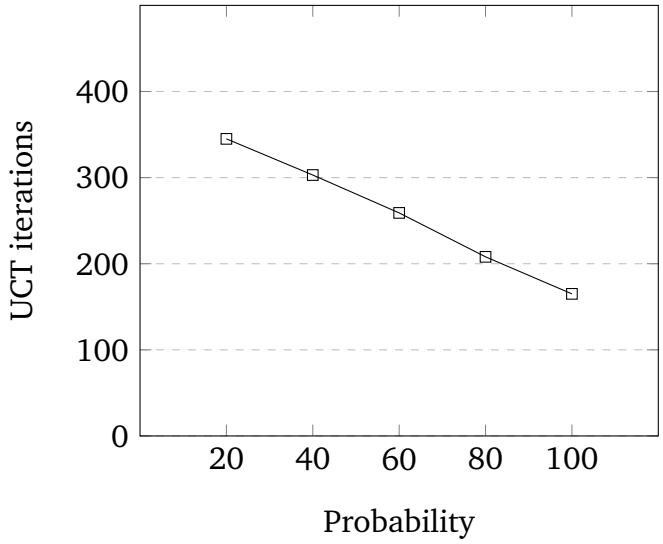


Figure 7: Probability dependence of UCT iterations within 10 seconds

function could affect the ability to explore the nodes. We chose MCC-SR-PR-p since it was the only approach which had a parameter that could have been configured without changing the evaluated position. All variants evaluated the position from Figure 2 within 10 seconds. The results are provided in Table 16.

Engine name	Number of performed iterations	Number of explored nodes
MCC-Solver	7838	1982366
MCC-SR- PR-20	345	40532
MCC-SR- PR-40	303	27647
MCC-SR- PR-60	259	19668
MCC-SR- PR-80	208	14152
MCC-SR	165	12967

Table 16: Number of performed UCT iterations and explored nodes within 10 seconds, evaluated position – the Légal trap

As we could notice, the number of performed iterations correlated to the achieved ranks which were provided in Section 5.4. However, the correlation was inverse, namely the more iterations a variant could do within the given time, the worse results it achieved. Comparing Tables 4 and 16, we could say that the probability of employing Stockfish's search function was linear dependent of performed UCT iterations (see Figure 7), that is, more UCT iterations would mean worse performance. Nonetheless, the difference of performed UCT iterations between MCC-Solver and other approaches was too big. This difference was probably occurred because Stockfish's time management couldn't work properly in case of many short calls to its search function. By adjusting the search depth of Stockfish we could have explored this problem in more detail. The used version of the Stockfish, however, didn't have the ability to adjust the search depth yet. We, thus, couldn't check the performance of this solution.

6 Conclusion and Future Work

Several modifications to MCTS were proposed recently in order to improve its performance. One of the most promising approaches consisted in employing informed minimax searches in the different phases of MCTS, which provided good results in the games of Othello, Catch the Lion and 6×6 Breakthrough. This thesis was aimed to test the similar approach in the game of chess. For this purpose, the search function of the state-of-art chess engine Stockfish has been embedded into MCC-Solver with the main focus on the rollout phase of MCTS.

Tests show that a naive approach would perform worse in fast blitz matches with an overall score of 32%. By letting the search function perform only in the selected phases of rollouts, we could achieve a significant improvement to our naive approach which resulted in the score of 77%. Another possible ways to improve the naive approach consisted in calling the search function only if there were either a predefined number of remaining pieces left on the board or if the game position was in a specific game phase. Both variants achieved very good scores against the naive approach – 83% for MCC-SR-32-25-R and 77% for MCC-SR-EG. By applying the search function in the selection and expansion phases we achieved a significant improvement over MCC-SR with an absolute win rate of 100%. In our main experiment we found out that employing the search function in the end game phase had achieved the best result compared to all other approaches. We let all proposed methods along with the baseline and except for the variant with bounded informed rollouts use the same computation time. Moreover, we explored how many short calls to Stockfish's search function could have affected the ability of our approaches to explore the nodes in the rollout phase. A variant which used the probability to call the search function was tested in order to help to understand, how far many short calls to Stockfish's search function could affect the ability of the purposed approaches to explore the nodes in the rollout phase.

As already mentioned, the naive approach performed worse against MCC-Solver in the 5 minutes blitz games. However, this approach has tackled the problem of search traps. Since both algorithms used a fixed computation time, an improvement of the naive approach could have been achieved by just increasing the computation time. A possible way to do this could be by parallelizing the rollouts or by adjusting the search depth. The last point could be done easy by employing the pure Minimax algorithm with an α-β pruning.

This thesis also show that using the search function in the selection and expansion phases gave performance boosts as well. Exploring the influence of informed searches in selection/expansion and backpropagation phases could be another possible direction for future work.

Recently, AlphaGo, a program that makes use of MCTS and convolutional neural networks, has beaten both European and World champions at the full-sized game of Go [SHM+16], [DAL]. We assume that applying the same method to the game of chess might be the most interesting direction for future work.

7 Appendix

1. Nf3 Nf6 2. Nc3 Nc6 3. d4 d5 4. Bf4 e6 5. e3 Bb4 6. Bd3 Bd7 7. O-O O-O 8. Qd2 Bd6 9. e4 dxe4 10. Nxe4 Nxe4 11. Bxe4 f5 12. Bd3 Bxf4 13. Qxf4 Rf6 14. c3 Ne7 15. Ne5 Ng6 16. Qe3 Nxe5 17. dxe5 Rh6 18. Rfd1 Qh4 19. h3 Bc6 20. f3 Rg6 21. Kh1 Qg3 22. Rd2 Rd8 23. Rad1 Kf7 24. Bc4 0-1

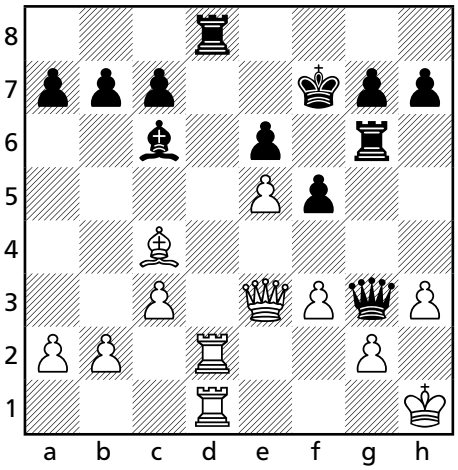


Figure 8: Game 11 from the match MCC-SR-EG vs. Stockfish

1. Nf3 Nf6 2. d4 Nc6 3. Nc3 d5 4. e3 Bf5 5. Bd3 Ne4 6. Bxe4 dxe4 7. Nh4 g6 8. Nxf5 gxf5 9. O-O e6 10. f3 exf3 11. Qxf3 Rg8 12. g3 Qg5 13. Qf2 O-O-O 14. e4 Qh5 15. d5 Ne5 16. Qxa7 Nf3+ 17. Rxf3 Qxf3 18. exf5 exd5 19. Be3 Bd6 20. Bf2 c6 21. Na4 d4 22. Nb6+ Kc7 23. Nc4 Kc8 24. Nxd6+ Rxd6 25. Qa8+ Kc7 26. Qxg8 Qxf5 27. Re1 Qxc2 28. Qxf7+ Kb6 29. Qf4 Qc5 30. Re4 Qd5 1-0

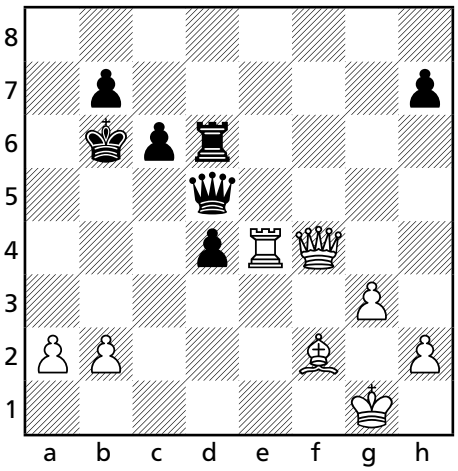


Figure 9: Game 12 from the match MCC-SR-EG vs. Stockfish

8 References

References

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, 2002.

[AHH10] B. Arneson, R. B. Hayward, and P. Henderson. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, Dec 2010.

[Are12] Oleg Arenz. Monter carlo chess. Bachelor thesis, Knowledge Engineering Group, TU Darmstadt, 2012.

[Bai15] Hendrik Baier. *Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains*. PhD thesis, Maastricht University, Maastricht, The Netherlands, 2015.

[BK88] John Birmingham and Peter Kent. *Tree-Searching and Tree-Pruning Techniques*, pages 123–128. Springer New York, New York, NY, 1988.

[BW12] Hendrik Baier and Mark H. M. Winands. *Time Management for Monte-Carlo Tree Search in Go*, pages 39–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[BW13] H. Baier and M. H. M. Winands. Monte-carlo tree search and minimax hybrids. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8, Aug 2013.

[CCP] Computer chess programming. https://verhelst.home.xs4all.nl/chess/search.html. Accessed: 28.08.2016.

[Cou07] Rémi Coulom. *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*, pages 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[CSB+06] G. Chaslot, J. Saito, B. Bouzy, J. Uiterwijk, and H. Van Den Herik. Monte-carlo strategies for computer go. *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91+, 2006.

[CWvdH08] Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. *Parallel Monte-Carlo Tree Search*, pages 60–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[DAL] Deepmind alphago. https://deepmind.com/alpha-go. Accessed: 28.08.2016.

[DTW12] N.G.P. Den Teuling and M.H.M. Winands. Monte-carlo tree search for the simultaneous move game tron. *Computer Games Workshop at ECAI 2012*, pages 126–141, 2012.

[EH61] Daniel Edwards and Timothy Hart. The alpha-beta heuristic. Technical report, Cambridge, MA, USA, 1961.

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. Technical report, INRIA, 2006.

[Har88] Larry Harris. *The Heuristic Search and The Game of Chess. A Study of Quiescence, Sacrifices, and Plan Oriented Play*, pages 136–142. Springer New York, New York, NY, 1988.

[Hei98] Ernst A. Heinz. *Extended Futility Pruning*. 1998.

[JC14] Nicolas Jouandeau and Tristan Cazenave. *Monte-Carlo Tree Reductions for Stochastic Games*, pages 228–238. Springer International Publishing, Cham, 2014.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[LH14] Richard Lorentz and Therese Horey. *Programming Breakthrough*, pages 49–59. Springer International Publishing, Cham, 2014.

[Lor11] Richard J. Lorentz. *Improving Monte–Carlo Tree Search in Havannah*, pages 105–115. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[MU49] N Metropolis and S Ulam. The monte carlo method. *Journal of the American statistical Association*, 44(247):335–341, 1949.

[Neu28] J. von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

[NW12] J. (Pim) A. M. Nijssen and Mark H. M. Winands. *Playout Search for Monte-Carlo Tree Search in Multi-player Games*, pages 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[PSPME12] P. Perick, D. L. St-Pierre, F. Maes, and D. Ernst. Comparison of different selection strategies in monte-carlo tree search for the game of tron. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 242–249, Sept 2012.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[RSS10] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On adversarial search spaces and sampling-based planning, 2010.

[RT10] Arpad Rimmel and Fabien Teytaud. *Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search*, pages 201–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Sco69] John J. Scott. A chess–playing program. *Machine Intelligence*, 4:255–265, 1969.

[Sha50] Claude E. Shannon. *Programming a Computer for Playing Chess*. 1950.

[SHM+16] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[Smi95] Warren Smith. Fixed point for negamaxing probability distributions on regular trees, 1995.

[STG10] Y. Sato, D. Takahashi, and Reijer Grimbergen. A shogi program based on monte-carlo tree search. *ICGA Journal*, 33(2):80–92, 2010.

[SWU12] Jan A. Stankiewicz, Mark H. M. Winands, and Jos W. H. M. Uiterwijk. *Monte-Carlo Tree Search Enhancements for Havannah*, pages 60–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[WBS08] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. *Monte-Carlo Tree Search Solver*, pages 25–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[XL09] F. Xie and Z. Liu. Backpropagation modification in monte-carlo game tree search. In *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, volume 2, pages 125–128, Nov 2009.

[XLW+10] Fan Xie, Zhiqing Liu, Yu Wang, Wenhao Huang, and Shuo Wang. *Systematic Improvement of Monte-Carlo Tree Search with Self-generated Neural-Networks Controllers*, pages 228–231. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.