Monte-Carlo Tree Search and Minimax Hybrids with Heuristic Evaluation Functions

Hendrik Baier and Mark H.M. Winands

Games and AI Group, Department of Knowledge Engineering
Faculty of Humanities and Sciences, Maastricht University
Maastricht, The Netherlands
{hendrik.baier,m.winands}@maastrichtuniversity.nl

Abstract. *Monte-Carlo Tree Search* (MCTS) has been found to play suboptimally in some tactical domains due to its highly selective search, focusing only on the most promising moves. In order to combine the strategic strength of MCTS and the tactical strength of minimax, *MCTS-minimax hybrids* have been introduced, embedding shallow minimax searches into the MCTS framework. Their results have been promising even without making use of domain knowledge such as heuristic evaluation functions. This paper continues this line of research for the case where evaluation functions are available. Three different approaches are considered, employing minimax with an evaluation function in the rollout phase of MCTS, as a replacement for the rollout phase, and as a node prior to bias move selection. The latter two approaches are newly proposed. The MCTS-minimax hybrids are tested and compared to their counterparts using evaluation functions without minimax in the domains of Othello, Breakthrough, and Catch the Lion. Results showed that introducing minimax search is effective for heuristic node priors in Othello and Catch the Lion. The MCTS-minimax hybrids are also found to work well in combination with each other. For their basic implementation in this investigative study, the effective branching factor of a domain is identified as a limiting factor of the hybrid's performance.

1 Introduction

Monte-Carlo Tree Search (MCTS) [7, 13] is a sampling-based tree search algorithm using the average result of Monte-Carlo simulations as state evaluations. It selectively samples promising moves instead of taking all legal moves into account like traditional minimax search. This leads to better performance in many large search spaces with high branching factors. MCTS also uses Monte-Carlo simulations of entire games, which often allows it to take long-term effects of moves better into account than minimax. If exploration and exploitation are traded off appropriately, MCTS asymptotically converges to the optimal policy [13], while providing approximations at any time.

While MCTS has shown considerable success in a variety of domains [4], it is still inferior to minimax search with alpha-beta pruning [12] in certain

games such as Chess and (International) Checkers. Part of the reason could be the selectivity of MCTS, its focusing on only the most promising lines of play. In tactical games such as Chess, a large number of traps exist in the search space [19]. These require precise play to avoid immediate loss, and the selective sampling of MCTS based on average simulation outcomes can easily miss or underestimate an important move.

In previous work [2], the tactical strength of minimax has been combined with the strategic and positional understanding of MCTS in *MCTS-minimax hybrids*, integrating shallow-depth minimax searches into the MCTS framework. These hybrids have shown promising results in tactical domains, despite being independent of a heuristic evaluation function for non-terminal states as typically needed by minimax. In this follow-up paper, we focus on the common case where evaluation functions are available. State evaluations can either result from simple evaluation function calls, or be backpropagated from shallow embedded minimax searches using the same evaluation function. This integration of minimax into MCTS accepts longer computation times in favor of typically more accurate state evaluations.

Three different approaches for integrating domain knowledge into MCTS are considered in this paper. The first approach uses state evaluations to choose rollout moves. The second approach uses state evaluations to terminate rollouts early. The third approach uses state evaluations to bias the selection of moves in the MCTS tree. Only in the first case, minimax has been applied before. The use of minimax for the other two approaches is newly proposed in the form described here.

This paper is structured as follows. Section 2 gives some background on MCTS as the baseline algorithm of this paper. Section 3 provides a brief overview of related work on the relative strengths of minimax and MCTS, on algorithms combining features of MCTS and minimax, and on using MCTS with heuristics. Section 4 outlines three different methods for incorporating heuristic evaluations into the MCTS framework, and presents variants using shallow-depth minimax searches for each of these. Two of these MCTS-minimax hybrids are newly proposed in this work. Section 5 shows experimental results of the MCTS-minimax hybrids in the test domains of Othello, Breakthrough, and Catch the Lion. Section 6 concludes and suggests future research.

2 Background

Monte-Carlo Tree Search (MCTS) is the underlying framework of the algorithms in this paper. MCTS works by repeating the following four-phase loop until computation time runs out [5]. The root of the tree represents the current state of the game. Each iteration of the loop represents one simulated game.

Phase one: *selection*. The tree is traversed starting from the root, choosing the move to sample from each state with the help of a selection policy. This policy should balance the exploitation of states with high value estimates and

the exploration of states with uncertain value estimates. In this paper UCB1-TUNED [1] is used as a selection policy.

Phase two: *expansion*. When the selection policy leaves the tree by sampling an unseen move, one or more of its successors are added to the tree. In this paper, we always add the one successor chosen in the current iteration.

Phase three: *rollout*. A rollout (also called *playout*) policy plays the simulated game to its end, starting from the state represented by the newly added node. MCTS converges to the optimal move in the limit even when rollout moves are chosen randomly.

Phase four: *backpropagation*. The value estimates of all states traversed during the simulation are updated with the result of the finished game.

Many variants and extensions of this framework have been proposed in the literature [4]. In this paper, we are using MCTS with the *MCTS-Solver* extension [28] as the baseline algorithm. MCTS-Solver is able to backpropagate not only regular simulation results such as losses and wins, but also game-theoretic values such as proven losses and proven wins whenever the search tree encounters a terminal state. The basic idea is marking a move as a proven loss if the opponent has a winning move from the resulting position, and marking a move as a proven win if the opponent has only losing moves from the resulting position. This avoids wasting time on the re-sampling of game states whose values are already known.

3 Related Work

Several papers by Ramanujan *et al.* [19,21,22] have studied search space properties that influence the performance of MCTS relative to minimax search. In [19], *shallow traps* were identified as a feature of search spaces in which MCTS performs poorly, in particular Chess. A *level-k search trap* was informally defined as the possibility of a player to choose an unfortunate move which leads to a winning strategy for the opponent with a depth of at most k plies. Such traps turned out to be frequent in Chess compared to for example Go. A synthetic tree model allowed the study of MCTS performance at different densities of traps in the search space in [21].

Finnsson and Björnsson [8] found a similar problem to traps, named *optimistic moves*. These are weak moves with relatively easy refutations by the opponent which take MCTS a surprisingly long time to find. In the same paper, the *progression* property was found to be advantageous for MCTS, i.e. the property of a game to progress naturally towards its end with every move made, as compared to games whose ends can be easily delayed or dragged out.

Clune [6] compared the performance of minimax with alpha-beta pruning and MCTS in General Game Playing. He found a stable and accurate evaluation function as well as a relatively low branching factor to give minimax an advantage over MCTS. In this paper, branching factor, evaluation accuracy and trap density help us to understand some of the observed effects.

Previous work on developing algorithms influenced by both MCTS and minimax has taken two principal approaches. On the one hand, one can extract

individual features of minimax such as minimax-style backups and integrate them into MCTS. This approach was chosen e.g. in [22], where the algorithm $UCTMAX_H$ replaces MCTS rollouts with heuristic evaluations and classic averaging MCTS backups with minimaxing backups. In *implicit minimax backups* [14], both minimaxing backups of heuristic evaluations and averaging backups of rollout returns are managed simultaneously. On the other hand, one can nest minimax searches into MCTS searches. This is the approach taken in [2] and this paper.

Various different techniques for integrating domain knowledge into the Monte-Carlo rollouts have been proposed in the literature. The idea of improving rollouts with the help of heuristic knowledge has first been applied to games in [3]. It is now used by state-of-the-art programs in virtually all domains. Shallow minimax in every step of the rollout phase has been proposed as well, e.g. a 1-ply search in [17] for the game of Havannah, or a 2-ply search for Lines of Action [27], Chess [20], and multi-player games [18]. Similar techniques are considered in Subsection 4.1.

The idea of stopping rollouts before the end of the game and backpropagating results on the basis of heuristic knowledge has been explored in Amazons [15], Lines of Action [26], and Breakthrough [16]. A similar method is considered in Subsection 4.2, where we also introduce a hybrid algorithm replacing the evaluation function with a minimax call. Our methods are different from [15] and [26] as we backpropagate the actual heuristic values instead of rounding them to losses or wins. They are also different from [26] as we backpropagate heuristic values after a fixed number of rollout moves, regardless of whether they reach a threshold of certainty.

The idea of biasing the selection policy with heuristic knowledge has been introduced in [9] and [5] for the game of Go. Our implementation is similar to [9] as we initialize tree nodes with knowledge in the form of virtual wins and losses. We also propose a hybrid using minimax returns instead of simple evaluation returns in Subsection 4.3.

This paper represents the continuation of earlier work on MCTS-minimax hybrids [2]. These hybrids MCTS-MR, MCTS-MS, and MCTS-MB have the advantage of being independent of domain knowledge. However, their inability to evaluate non-terminal states makes them ineffective in games with very few or no terminal states throughout the search space, such as the game of Othello. Furthermore, some form of domain knowledge is often available in practice, and it is an interesting question how to use it to maximal effect.

4 Hybrid Algorithms

This section describes the three different approaches for employing heuristic knowledge within MCTS that we explore in this work. For each approach, a variant using simple evaluation function calls and a hybrid variant using shallow minimax searches is considered. Two of the three hybrids are newly proposed in the form described here.

4.1 MCTS with Informed Rollouts (MCTS-IR)

The convergence of MCTS to the optimal policy is guaranteed even with uniformly random move choices in the rollouts. However, more informed rollout policies can greatly improve performance [10]. When a heuristic evaluation function is available, it can be used in every rollout step to compare the states each legal move would lead to, and choose the most promising one. Instead of choosing this *greedy* move, it is effective in some domains to choose a uniformly random move with a low probability ϵ, so as to avoid determinism and preserve diversity in the rollouts. Our implementation additionally ensures non-deterministic behavior even for $\epsilon = 0$ by picking moves with equal values at random both in the selection and in the rollout phase of MCTS. The resulting rollout policy is typically called *ϵ-greedy* [25]. In the context of this work, we call this approach *MCTS-IR-E* (MCTS with informed rollouts using an evaluation function).

The depth-one lookahead of an ϵ-greedy policy can be extended in a natural way to a depth-d minimax search for every rollout move [18,27]. We use a random move ordering in minimax as well in order to preserve non-determinism. In contrast to [27] and [18] where several enhancements such as move ordering, k-best pruning (not searching all legal moves), and killer moves were added to alpha-beta, we only use basic alpha-beta search. We are interested in its performance before introducing additional improvements, especially since our test domains have smaller branching factors than e.g. the games Lines of Action (around 30) or Chinese Checkers (around 25-30) used in [27] and [18], respectively. Using a depth-d minimax search for every rollout move aims at stronger move choices in the rollouts, which make rollout returns more accurate and can therefore help to guide the growth of the MCTS tree. We call this approach *MCTS-IR-M* (MCTS with informed rollouts using minimax).

4.2 MCTS with Informed Cutoffs (MCTS-IC)

The idea of rollout cutoffs is an early termination of the rollout in case the rollout winner, or the player who is at an advantage, can be reasonably well predicted with the help of an evaluation function. The statistical noise introduced by further rollout moves can then be avoided by stopping the rollout, evaluating the current state of the simulation, and backpropagating the evaluation result instead of the result of a full rollout to the end of the game [15, 26]. If on average, the evaluation function is computationally cheaper than playing out the rest of the rollout, this method can also result in an increased sampling speed as measured in rollouts per second. A fixed number m of rollout moves can be played before evaluating in order to introduce more non-determinism and get more diverse simulation returns. If $m = 0$, the evaluation function is called directly at the newly expanded node of the tree. As in MCTS-IR, our MCTS-IC implementation avoids deterministic gameplay through randomly choosing among equally valued moves in the selection policy. We scale all evaluation values to $[0, 1]$. We do not round the evaluation function values to wins or losses as proposed in [15], nor do we consider the variant with dynamic m and evaluation

function thresholds proposed in [26]. In the following, we call this approach *MCTS-IC-E* (MCTS with informed cutoffs using an evaluation function).

We propose an extension of this method using a depth-d minimax search at cutoff time in order to determine the value to be backpropagated. In contrast to the integrated approach taken in [27], we do not assume MCTS-IR-M as rollout policy and backpropagate a win or a loss whenever the searches of this policy return a value above or below two given thresholds. Instead, we play rollout moves with an arbitrary policy (uniformly random unless specified otherwise), call minimax when a fixed number of rollout moves has been reached, and backpropagate the heuristic value returned by this search. Like MCTS-IR-M, this strategy tries to backpropagate more accurate simulation returns, but by computing them directly instead of playing out the simulation. We call this approach *MCTS-IC-M* (MCTS with informed cutoffs using minimax).

4.3 MCTS with Informed Priors (MCTS-IP)

Node priors [9] represent one method for supporting the selection policy of MCTS with heuristic information. When a new node is added to the tree, or after it has been visited n times, the heuristic evaluation of the corresponding state is stored in this node. This is done in the form of virtual wins and virtual losses, weighted by a prior weight parameter w. For example, if the evaluation value is 0.6 and the weight is 100, 60 wins and 40 losses are stored in the node at hand. We assume evaluation values in $[0, 1]$. Since heuristic evaluations are typically more reliable than the MCTS value estimates resulting from only a few samples, this prior helps to guide tree growth into a promising direction. If the node is visited frequently however, the influence of the prior progressively decreases over time, as the virtual rollout returns represent a smaller and smaller percentage of the total rollout returns stored in the node. Thus, MCTS rollouts progressively override the heuristic evaluation. We call this approach *MCTS-IP-E* (MCTS with informed priors using an evaluation function) in this paper.

We propose to extend this technique with a depth-d minimax search in order to compute the prior value to be stored. It aims at guiding the selection policy through more accurate prior information in the MCTS tree. We call this approach *MCTS-IP-M* (MCTS with informed priors using minimax).

5 Experimental Results

We tested the MCTS-minimax hybrids with heuristic evaluation functions in three different two-player zero-sum games: *Othello*, *Breakthrough*, and *Catch the Lion*. In all experimental conditions, we compared the hybrids as well as their counterparts using heuristics without minimax against regular MCTS-Solver as the baseline. Rollouts were uniformly random unless specified otherwise. Optimal MCTS parameters such as the exploration factor C were determined once for MCTS-Solver in each game and then kept constant for both MCTS-Solver and the MCTS-minimax hybrids during testing. C was 0.7 in Othello and Catch

the Lion, and 0.8 in Breakthrough. Draws, which are possible in Othello, were counted as half a win for both players. We used minimax with alpha-beta pruning, but no other search enhancements. Computation time was 1 second per move.

5.1 Games

This section outlines the rules of the three test domains, and the heuristic board evaluation functions used for each of them. The evaluation function from the point of view of the current player is always her total score minus her opponent's total score, normalized to $[0, 1]$ as a final step.

Othello. The game of Othello is played on an 8×8 board. It starts with four discs on the board, two white discs on d5 and e4 and two black discs on d4 and e5. Each disc has a black side and a white side, with the side facing up indicating the player the disc currently belongs to. The two players alternatingly place a disc on the board, in such a way that between the newly placed disc and another disc of the moving player there is an uninterrupted horizontal, vertical or diagonal line of one or more discs of the opponent. All these discs are then turned over, changing their color to the moving player's side, and the turn goes to the other player. If there is no legal move for a player, she has to pass. If both players have to pass or if the board is filled, the game ends. The game is won by the player who owns the most discs at the end.

The evaluation score we use for Othello first determines the number of *stable* discs for the player, i.e. discs that cannot change color anymore. For each stable disc of her color, the player receives 10 points. Afterwards, the number of legal moves for the player is added to her score.

Breakthrough. The variant of Breakthrough used in this work is played on a 6×6 board. The game was originally described as being played on a 7×7 board, but other sizes such as 8×8 are popular as well, and the 6×6 board preserves an interesting search space.

At the beginning of the game, White occupies the first two rows of the board, and Black occupies the last two rows of the board. The two players alternatingly move one of their pieces straight or diagonally forward. Two pieces cannot occupy the same square. However, players can capture the opponent's pieces by moving diagonally onto their square. The game is won by the player who succeeds first at advancing one piece to the home row of her opponent, i.e. reaching the first row as Black or reaching the last row as White.

The evaluation score we use for Breakthrough gives the player 3 points for each piece of her color. Additionally, each piece receives a location value depending on its row on the board. From the player's home row to the opponent's home row, these values are 10, 3, 6, 10, 15, and 21 points, respectively.

Catch the Lion. The game Catch the Lion is a simplified form of Shogi (see [23] for an MCTS approach to Shogi). It is included in this work as an example of chess-like games, which tend to be particularly difficult for MCTS [19].

The game is played on a 3×4 board. At the beginning of the game, each player has four pieces: a Lion in the center of her home row, a Giraffe to the right of the Lion, an Elephant to the left of the Lion, and a Chick in front of the Lion. The Chick can move one square forward, the Giraffe can move one square in the vertical and horizontal directions, the Elephant can move one square in the diagonal directions, and the Lion can move one square in any direction. During the game, the players alternatingly move one of their pieces. Pieces of the opponent can be captured. As in Shogi, they are removed from the board, but not from the game. Instead, they switch sides, and the player who captured them can later on drop them on any square of the board instead of moving one of her pieces. If the Chick reaches the home row of the opponent, it is promoted to a Chicken, now being able to move one square in any direction except for diagonally backwards. A captured Chicken, however, is demoted to a Chick again when dropped. The game is won by either capturing the opponent's Lion, or moving your own Lion to the home row of the opponent.

The evaluation score we use for Catch the Lion represents a weighted material sum for each player, where a Chick counts as 3 points, a Giraffe or Elephant as 5 points, and a Chicken as 6 points, regardless of whether they are on the board or captured by the player.

5.2 Game Properties

Two properties of the test domains can help with understanding the results presented in the following subsections. These properties are the *branching factor* and the *tacticality* of the games.

Branching factor. There are on average 15.5 legal moves available in Breakthrough, but only about 10 in Catch the Lion and 8 in Othello, measured in self-play games of the MCTS-Solver baseline. A higher branching factor makes the application of minimax searches potentially more difficult, especially when basic alpha-beta without enhancements is used as this paper.

Tacticality. The tacticality of a game can be formalized in different ways. [19] proposed the concept of search traps to explain the difficulties of MCTS in some domains such as Chess. This concept was taken up again in [2] to motivate the integration of minimax into MCTS. A tactical game is here understood as a game with a high density of terminal states throughout the search space, which can result in a higher risk of falling into traps especially for selective searches.

As a simple test for this property, MCTS-Solver played 1000 self-play games in all domains. After each move, we measured the number of traps at depth (up to) 3 for the player to move. The result was an average number of 3.7 level-3 traps in Catch the Lion (37% of all legal moves), 2.8 traps in Breakthrough

(18% of all legal moves), and only 0.1 traps in Othello (1.2% of all legal moves). Results were comparable for other trap depths. This indicates that Catch the Lion is the most tactical of the tested games, making the application of minimax searches potentially more useful.

5.3 Experiments with MCTS-IR

MCTS-IR-E was tested for $\epsilon \in \{0, 0.05, 0.1, 0.2, 0.5\}$. Each parameter setting played 1000 games in each domain against the baseline MCTS-Solver with uniformly random rollouts. Figures 1(a) to 1(c) show the results. The best-performing conditions used $\epsilon = 0.05$ in Othello and Catch the Lion, and $\epsilon = 0$ in Breakthrough. They were each tested in 2000 additional games against the baseline. The results were win rates of 79.9% in Othello, 75.4% in Breakthrough, and 96.8% in Catch the Lion. All of these are significantly stronger than the baseline ($p<0.001$).

MCTS-IR-M was tested for $d \in \{1, \ldots, 4\}$ with the optimal value of ϵ found for each domain in the MCTS-IR-E experiments. Each condition played 1000 games per domain against the baseline player. The results are presented in Figures 1(d) to 1(f). The most promising setting in all domains was $d = 1$. In an additional 2000 games against the baseline per domain, this setting achieved win rates of 73.9% in Othello, 65.7% in Breakthrough, and 96.5% in Catch the Lion. The difference to the baseline is significant in all domains ($p<0.001$).

In each domain, the best settings for MCTS-IR-E and MCTS-IR-M were then tested against each other in 2000 further games. The results for MCTS-IR-M were win rates of 37.1% in Othello, 35.3% in Breakthrough, and 47.9% in Catch the Lion. MCTS-IR-M is weaker than MCTS-IR-E in Othello and Breakthrough ($p<0.001$), while no significant difference could be shown in Catch the Lion. This shows that the incorporation of shallow alpha-beta searches into rollouts did not improve MCTS-IR in any of the domains at hand. Depth-1 minimax searches in MCTS-IR-M are functionally equivalent to MCTS-IR-E, but have some overhead in our implementation due to the recursive calls to a separate alpha-beta search algorithm. This results in the inferior performance.

Higher settings of d were not successful because deeper minimax searches in every rollout step require too much computational effort. In an additional set of 1000 games per domain, we compared MCTS-IR-E to MCTS-IR-M at 1000 rollouts per move, ignoring the time overhead of minimax. Here, MCTS-IR-M won 78.6% of games with $d = 2$ in Othello, 63.4% of games with $d = 2$ in Breakthrough, and 89.3% of games with $d = 3$ in Catch the Lion. All of these conditions are significantly stronger than MCTS-IR-E ($p<0.001$). This confirms MCTS-IR-M is suffering from its time overhead.

Interestingly, deeper minimax searches do not always guarantee better performance in MCTS-IR-M, even when ignoring time. While MCTS-IR-M with $d = 1$ won 50.4% ($\pm$3.1%) of 1000 games against MCTS-IR-E in Catch the Lion, $d = 2$ won only 38.0%—both at 1000 rollouts per move. In direct play against each other, MCTS-IR-M with $d = 2$ won 38.8% of 1000 games against MCTS-IR-M with $d = 1$. As standalone players however, a depth-2 minimax beat a

depth-1 minimax in 95.8% of 1000 games. Such cases where policies that are stronger as standalone players do not result in stronger play when integrated in MCTS rollouts have been observed before [2, 9, 24].

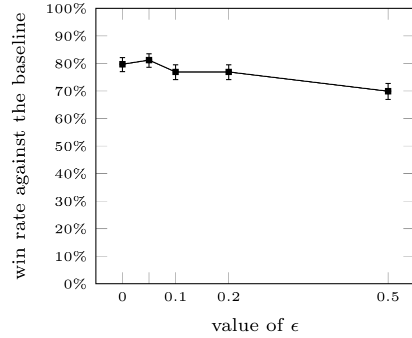
5.4 Experiments with MCTS-IC

MCTS-IC-E was tested for $m \in \{0, \ldots, 5\}$. 1000 games were played against the baseline MCTS-Solver per parameter setting in each domain. Figures 2(a) to 2(c) present the results. The most promising condition was $m = 0$ in all three domains. It was tested in 2000 additional games against the baseline. The results were win rates of 61.1% in Othello, 41.9% in Breakthrough, and 98.1% in Catch the Lion. This is significantly stronger than the baseline in Othello and Catch the Lion ($p<0.001$), but weaker in Breakthrough ($p<0.001$). The evaluation function in Breakthrough may not be accurate enough for MCTS to fully rely on it instead of rollouts. Testing higher values of m showed that as fewer and fewer rollouts are long enough to be cut off, MCTS-IC-E effectively turns into the baseline MCTS-Solver and also shows identical performance. Note that the parameter m can sometimes be sensitive to the opponents it is tuned against. In this paper, we tuned against regular MCTS-Solver only, and both MCTS-Solver and MCTS-IC used uniformly random rollouts.

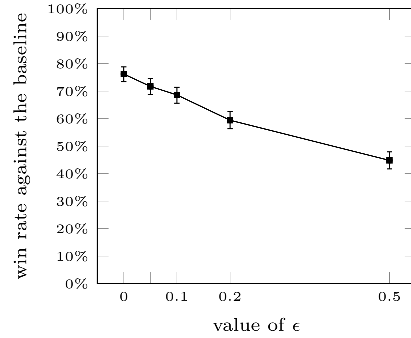
MCTS-IC-M was tested for all combinations of $m \in \{0, \ldots, 5\}$ and $d \in \{1, 2, 3\}$, with 1000 games each per domain. The results are shown in Figures 2(d) to 2(f). The best performance was achieved with $m = 0$ and $d = 2$ in Othello, $m = 4$ and $d = 1$ in Breakthrough, and $m = 1$ and $d = 2$ in Catch the Lion. Of an additional 2000 games against the baseline per domain, these settings won 62.4% in Othello, 32.4% in Breakthrough, and 99.6% in Catch the Lion. This is again significantly stronger than the baseline in Othello and Catch the Lion ($p<0.001$), but weaker in Breakthrough ($p<0.001$).

The best settings for MCTS-IC-E and MCTS-IC-M were also tested against each other in 2000 games per domain. Despite MCTS-IC-E and MCTS-IC-M not showing significantly different performance against the regular MCTS-Solver baseline in Othello and Catch the Lion, MCTS-IC-E won 73.1% of these games in Othello, 58.3% in Breakthrough, and 66.1% in Catch the Lion. All conditions are significantly superior to MCTS-IC-M ($p<0.001$). Thus, the integration of shallow alpha-beta searches into rollout cutoffs did not improve MCTS-IC in any of the tested domains either.

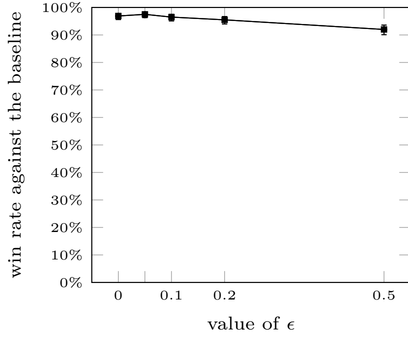
Just as for MCTS-IR, this is a problem of computational cost for the alpha-beta searches. We compared MCTS-IC-E with optimal parameter settings to MCTS-IC-M at equal rollouts per move instead of equal time in an additional set of experiments. Here, MCTS-IC-M won 65.7% of games in Othello at 10000 rollouts per move, 69.8% of games in Breakthrough at 6000 rollouts per move, and 86.8% of games in Catch the Lion at 2000 rollouts per move (the rollout numbers were chosen so as to achieve comparable times per move). The parameter settings were $m = 0$ and $d = 1$ in Othello, $m = 0$ and $d = 2$ in Breakthrough, and $m = 0$ and $d = 4$ in Catch the Lion. All conditions here are stronger than



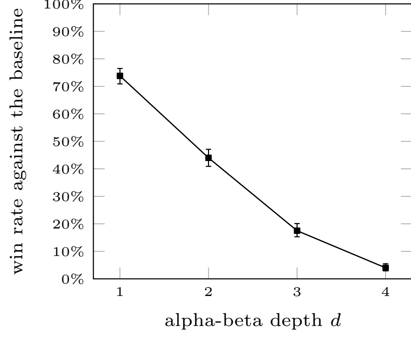
(a) Performance of MCTS-IR-E in Othello.



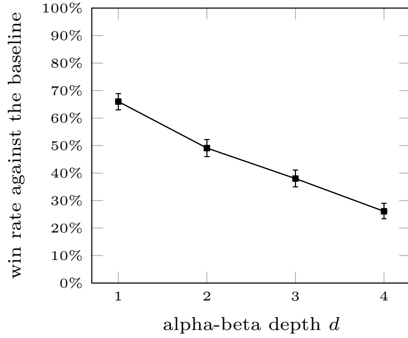
(b) Performance of MCTS-IR-E in Breakthrough.



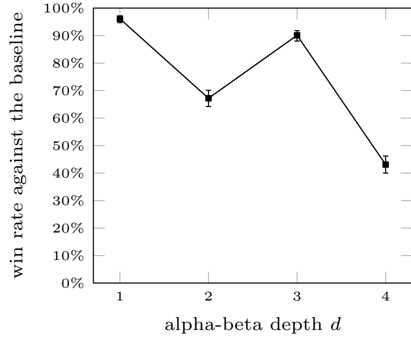
(c) Performance of MCTS-IR-E in Catch the Lion.



(d) Performance of MCTS-IR-M in Othello. For all conditions, $\epsilon = 0.05$.



(e) Performance of MCTS-IR-M in Breakthrough. For all conditions, $\epsilon = 0$.



(f) Performance of MCTS-IR-M in Catch the Lion. For all conditions, $\epsilon = 0.05$.

Fig. 1: Performance of MCTS-IR in Othello, Breakthrough and Catch the Lion.

MCTS-IC-E ($p<0.001$). This confirms that MCTS-IC-M is weaker than MCTS-IC-E due to its time overhead.

A seemingly paradoxical observation was made with MCTS-IC as well. In Breakthrough and Catch the Lion, the values returned by minimax searches are not always more effective for MCTS-IC than the values of simple static heuristics, even when time is ignored. In Catch the Lion for example, MCTS-IC-M with $m = 0$ and $d = 1$ won only 2.9% of 1000 test games against MCTS-IC-E with $m = 0$ (at 50000 rollouts per move). With $d = 2$, it won 34.3% (at 25000 rollouts per move). Even with $d = 3$, it won only 34.8% (at 6000 rollouts per move). Once more these results demonstrate that a stronger policy can lead to a weaker search when embedded in MCTS.

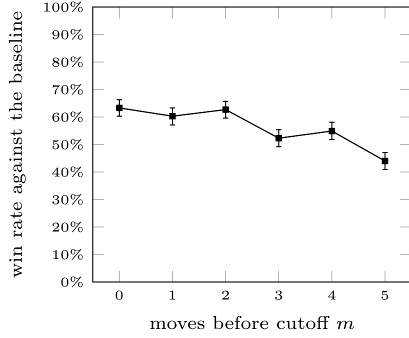
5.5 Experiments with MCTS-IP

MCTS-IP-E was tested for all combinations of $n \in \{0, 1, 2\}$ and $w \in \{50, 100, 250, 500, 1000, 2500, 5000\}$. Each condition played 1000 games per domain against the baseline player. The results are shown in Figures 3(a) to 3(c). The best-performing conditions were $n = 1$ and $w = 1000$ in Othello, $n = 1$ and $w = 2500$ in Breakthrough, and $n = 0$ and $w = 100$ in Catch the Lion. In 2000 additional games against the baseline, these conditions achieved win rates of 56.8% in Othello, 86.6% in Breakthrough, and 71.6% in Catch the Lion (all significantly stronger than the baseline with $p<0.001$).

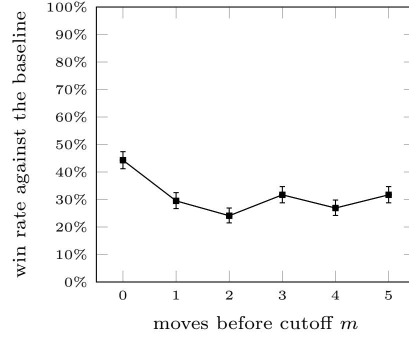
MCTS-IP-M was tested for all combinations of $n \in \{0, 1, 2, 5, 10, 25\}$, $w \in \{50, 100, 250, 500, 1000, 2500, 5000\}$, and $d \in \{1, \ldots, 5\}$ with 1000 games per condition in each domain. Figures 3(d) to 3(f) present the results, using the optimal setting of d for all domains. The most promising parameter values found in Othello were $n = 2$, $w = 5000$, and $d = 3$. In Breakthrough they were $n = 1$, $w = 1000$, and $d = 1$, and in Catch the Lion they were $n = 1$, $w = 2500$, and $d = 5$. Each of them played 2000 additional games against the baseline, winning 81.7% in Othello, 87.8% in Breakthrough, and 98.0% in Catch the Lion (all significantly stronger than the baseline with $p<0.001$).

The best settings for MCTS-IP-E and MCTS-IP-M subsequently played 2000 games against each other in all domains. MCTS-IP-M won 76.2% of these games in Othello, 97.6% in Catch the Lion, but only 36.4% in Breakthrough (all of the differences are significant with $p<0.001$). We can conclude that using shallow alpha-beta searches to compute node priors strongly improves MCTS-IP in Othello and Catch the Lion, but not in Breakthrough. This is once more a problem of time overhead due to the larger branching factor of Breakthrough. At 1000 rollouts per move, MCTS-IP-M with $n = 1$, $w = 1000$, and $d = 1$ won 91.1% of 1000 games against the best MCTS-IP-E setting in this domain.

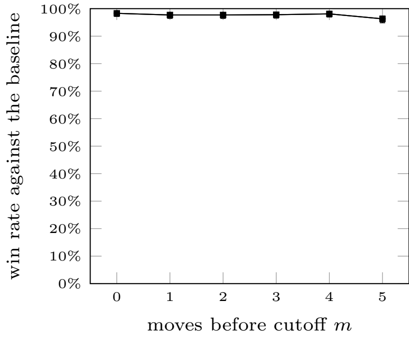
An interesting observation are the high weights assigned to the node priors in all domains. It seems that at least for uniformly random rollouts, best performance is achieved when rollout returns never override priors for the vast majority of nodes. They only differentiate between states that look equally promising for the evaluation functions used. The exception is MCTS-IP-E in Catch the Lion, where the static evaluations might be too unreliable to give them large weights



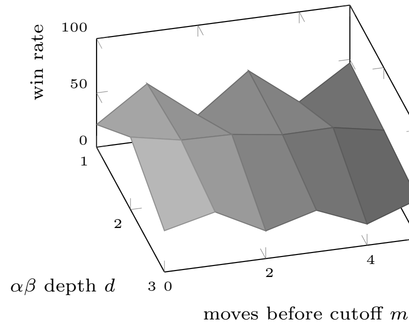
(a) Performance of MCTS-IC-E in Othello.



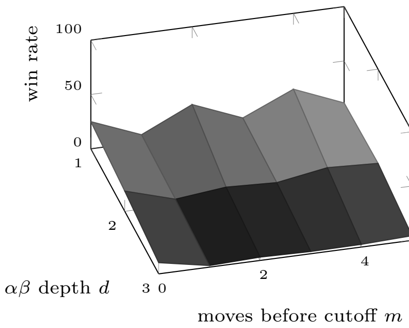
(b) Performance of MCTS-IC-E in Breakthrough.



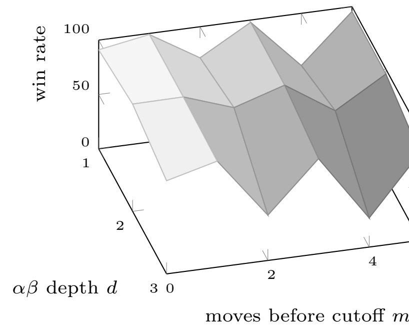
(c) Performance of MCTS-IC-E in Catch the Lion.



(d) Performance of MCTS-IC-M in Othello.



(e) Performance of MCTS-IC-M in Breakthrough.



(f) Performance of MCTS-IC-M in Catch the Lion.

Fig. 2: Performance of MCTS-IC in Othello, Breakthrough and Catch the Lion.

due to the tactical nature of the game. Exchanges of pieces can often lead to quick and drastic changes of the evaluation values. The quality of the priors in Catch the Lion improves drastically when minimax searches are introduced, justifying deeper searches ($d = 5$) than in the other tested domains despite the high computational cost. However, MCTS-IC still works better in this case, possibly because inaccurate evaluation results are only backpropagated once and are not stored to influence the selection policy for a longer time as in MCTS-IP. In Othello, minimax searches in combination with a seemingly less volatile evaluation function lead to MCTS-IP-M being the strongest hybrid tested in this paper.

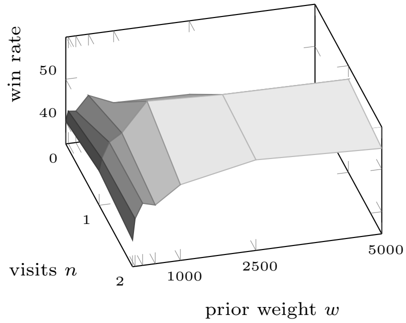
The effect of stronger policies resulting in weaker performance when integrated into MCTS can be found in MCTS-IP just as in MCTS-IR and MCTS-IC. In Breakthrough for example, MCTS-IP-M with $n = 1$, $w = 1000$, and $d = 2$ won only 83.4% of 1000 games against the strongest MCTS-IP-E setting, compared to 91.1% with $n = 1$, $w = 1000$, and $d = 1$—both at 1000 rollouts per move. The difference is significant ($p<0.001$). As standalone players however, depth-2 minimax won 80.2% of 1000 games against depth-1 minimax in the Breakthrough experiments.

5.6 Comparison of Algorithms

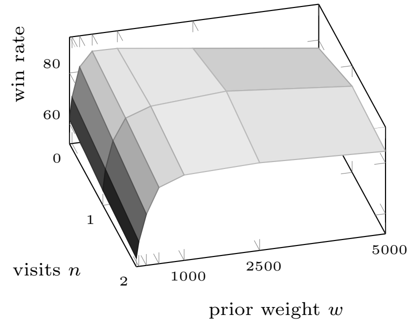
Sections 5.3 to 5.5 showed the performance of MCTS-IR, MCTS-IC and MCTS-IP against the baseline MCTS-Solver player. We also tested the best-performing variants of these algorithms against each other. In each condition, 2000 games were played. Figures 4(a) to 4(c) present the results. MCTS-IP-M is strongest in Othello, MCTS-IP-E is strongest in Breakthrough, and MCTS-IC-E is strongest in Catch the Lion.

5.7 Combination of Algorithms

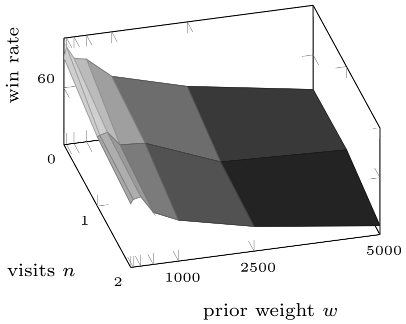
Subsections 5.3 to 5.6 showed the performance of MCTS-IR, MCTS-IC and MCTS-IP in isolation. In order to get an indication whether the different methods of applying heuristic knowledge can successfully be combined, we conducted the following experiments. In Othello, the best-performing algorithm MCTS-IP-M was combined with MCTS-IR-E. In Breakthrough, the best-performing algorithm MCTS-IP-E was combined with MCTS-IR-E. In Catch the Lion, it is not possible to combine the best-performing algorithm MCTS-IC-E with MCTS-IR-E, because with the optimal setting $m = 0$ MCTS-IC-E leaves no rollout moves to be chosen by an informed rollout policy. Therefore, MCTS-IP-M was combined with MCTS-IR-E instead. 2000 games were played in each condition. The results are shown in Figures 4(d) to 4(f). Applying the same knowledge both in the form of node priors and in the form of ϵ-greedy rollouts leads to stronger play in all three domains than using priors alone. In fact, such combinations are the overall strongest players tested in this paper even without being systematically optimized. In Othello, the combination MCTS-IP-M-IR-E won 55.2% of 2000 games against the strongest individual algorithm MCTS-IP-M (stronger with $p=0.001$). In Breakthrough, the combination MCTS-IP-E-IR-E won 53.9%



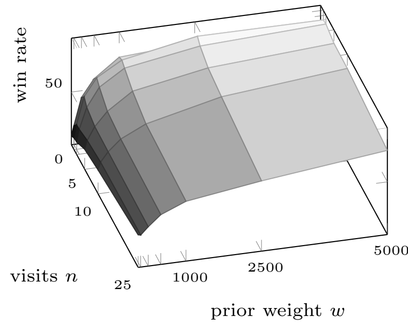
(a) Performance of MCTS-IP-E in Othello.



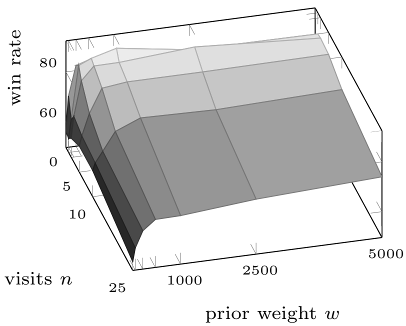
(b) Performance of MCTS-IP-E in Breakthrough.



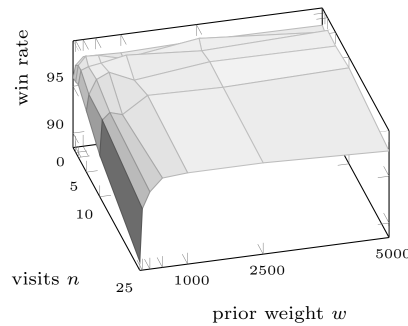
(c) Performance of MCTS-IP-E in Catch the Lion.



(d) Performance of MCTS-IP-M in Othello. For all conditions, $d = 3$.



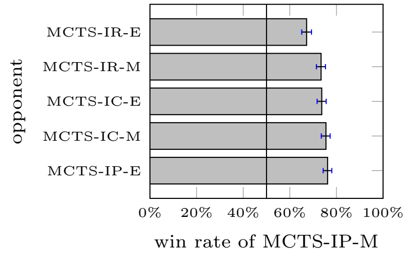
(e) Performance of MCTS-IP-M in Breakthrough. For all conditions, $d = 1$.



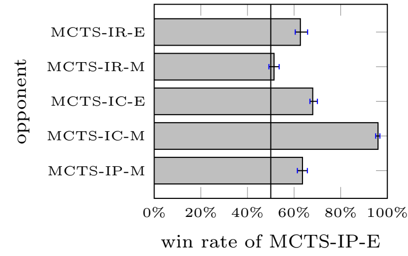
(f) Performance of MCTS-IP-M in Catch the Lion. For all conditions, $d = 5$.

Fig. 3: Performance of MCTS-IP in Othello, Breakthrough and Catch the Lion.

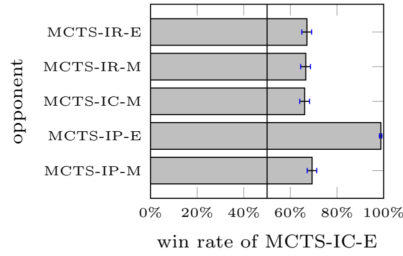
against the best-performing algorithm MCTS-IP-E (stronger with p<0.05). In Catch the Lion, the combination MCTS-IP-M-IR-E with $n = 1$, $w = 2500$, and $d = 4$ won 61.1% of 2000 games against the strongest algorithm MCTS-IC-E (stronger with p<0.001, not shown in Figure 4(f)).



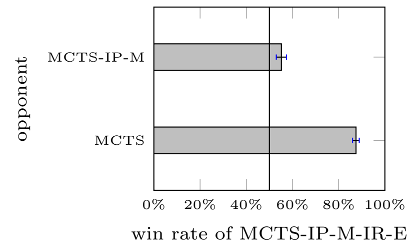
(a) Performance of MCTS-IP-M against the other hybrids in Othello.



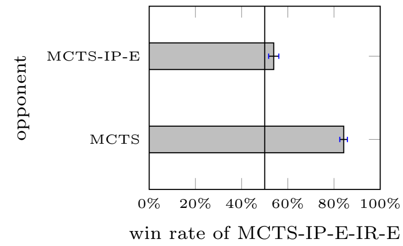
(b) Performance of MCTS-IP-E against the other hybrids in Breakthrough.



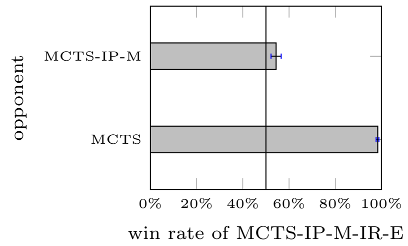
(c) Performance of MCTS-IC-E against the other hybrids in Catch the Lion.



(d) Performance of MCTS-IP-M combined with MCTS-IR-E in Othello.



(e) Performance of MCTS-IP-E combined with MCTS-IR-E in Breakthrough.



(f) Performance of MCTS-IP-M combined with MCTS-IR-E in Catch the Lion.

Fig. 4: Comparisons and combinations of the MCTS-minimax hybrids.

6 Conclusion and Future Research

In this paper, we considered three approaches for integrating heuristic state evaluation functions into MCTS. MCTS-IR uses heuristic knowledge to improve the rollout policy. MCTS-IC uses heuristic knowledge to terminate rollouts early. MCTS-IP uses heuristic knowledge as prior for tree nodes. In all three approaches, we also examined the computation of state evaluations with shallow-depth minimax searches using the same heuristic knowledge. This has only been done for MCTS-IR before.

Experimental results in the domains of Othello, Breakthrough and Catch the Lion showed that the best individual players tested in Othello and Breakthrough make use of priors in order to combine heuristic information with rollout returns. Because of the different branching factors, computing these priors works best by embedding shallow minimax searches in Othello, and by a simple evaluation function call in Breakthrough. In Catch the Lion, random rollouts may too often return inaccurate results due to the tacticality and possibly also due to the non-converging nature of the domain. Replacing these rollouts with the evaluation function turned out to be the most successful of the individually tested approaches. Preliminary experiments with combining the different approaches showed that in both Othello and Catch the Lion, using minimax to compute node priors and applying simple ϵ-greedy rollouts resulted in the overall most successful players tested in this paper.

The fact that some combinations of algorithms play at a higher level than the algorithms in isolation may mean we have not yet found a way to fully and optimally exploit our heuristic knowledge. This is a first direction for future research.

Second, differences between test domains such as their density of terminal states, their density of hard and soft traps [20], or their progression property [8] could be studied in order to understand the behavior of MCTS-minimax hybrids. Artificial game trees could be a valuable tool to separate the effects of individual properties.

Third, all three approaches for using heuristic knowledge have shown cases where embedded minimax searches did not lead to stronger MCTS play than shallower minimax searches or even simple evaluation function calls. This phenomenon has only been observed in MCTS-IR before and deserves further study.

Finally, the main problem of MCTS-minimax hybrids seems to be their sensitivity to the branching factor of the domain. This explains their weak performance in Breakthrough. However, the minimax implementation used in this paper was a simple, unenhanced alpha-beta search. An improved implementation with e.g. static move ordering, k-best pruning, and killer moves has been shown to allow for successful MCTS-IR-M even in Lines of Action, a domain with an average branching factor twice as high as 6×6 Breakthrough [26]. These techniques could drastically increase the branching factor for which all MCTS-minimax hybrids are viable.

Acknowledgment. This work is funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-Time Analysis of the Multiarmed Bandit Problem. Machine Learning 47(2-3), 235–256 (2002)
2. Baier, H., Winands, M.H.M.: Monte-Carlo Tree Search and Minimax Hybrids. In: 2013 IEEE Conference on Computational Intelligence and Games, CIG 2013. pp. 129–136 (2013)
3. Bouzy, B.: Associating Domain-Dependent Knowledge and Monte Carlo Approaches within a Go Program. Information Sciences 175(4), 247–257 (2005)
4. Browne, C., Powley, E.J., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games 4(1), 1–43 (2012)
5. Chaslot, G.M.J.B., Winands, M.H.M., van den Herik, H.J., Uiterwijk, J.W.H.M., Bouzy, B.: Progressive Strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation 4(3), 343–357 (2008)
6. Clune, J.E.: Heuristic Evaluation Functions for General Game Playing. Ph.D. thesis, University of California, Los Angeles, USA (2008)
7. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) 5th International Conference on Computers and Games (CG 2006). Revised Papers. Lecture Notes in Computer Science, vol. 4630, pp. 72–83. Springer (2007)
8. Finnsson, H., Björnsson, Y.: Game-Tree Properties and MCTS Performance. In: IJCAI'11 Workshop on General Intelligence in Game Playing Agents (GIGA'11). pp. 23–30 (2011)
9. Gelly, S., Silver, D.: Combining Online and Offline Knowledge in UCT. In: Ghahramani, Z. (ed.) 24th International Conference on Machine Learning, ICML 2007. ACM International Conference Proceeding Series, vol. 227, pp. 273–280 (2007)
10. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go. Tech. rep., HAL - CCSd - CNRS, France (2006)
11. van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.): Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5131. Springer (2008)
12. Knuth, D.E., Moore, R.W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6(4), 293–326 (1975)
13. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) 17th European Conference on Machine Learning, ECML 2006. Lecture Notes in Computer Science, vol. 4212, pp. 282–293. Springer (2006)
14. Lanctot, M., Winands, M.H.M., Pepels, T., Sturtevant, N.R.: Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. In: 2014 IEEE Conference on Computational Intelligence and Games, CIG 2014. pp. 341–348 (2014)

15. Lorentz, R.J.: Amazons Discover Monte-Carlo. In: van den Herik et al. [11], pp. 13–24
16. Lorentz, R.J., Horey, T.: Programming Breakthrough. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) 8th International Conference on Computers and Games, CG 2013. Lecture Notes in Computer Science, vol. 8427, pp. 49–59. Springer (2014)
17. Lorentz, R.J.: Experiments with Monte-Carlo Tree Search in the Game of Havannah. ICGA Journal 34(3), 140–149 (2011)
18. Nijssen, J.A.M., Winands, M.H.M.: Playout Search for Monte-Carlo Tree Search in Multi-player Games. In: van den Herik, H.J., Plaat, A. (eds.) 13th International Conference on Advances in Computer Games, ACG 2011. Lecture Notes in Computer Science, vol. 7168, pp. 72–83. Springer (2011)
19. Ramanujan, R., Sabharwal, A., Selman, B.: On Adversarial Search Spaces and Sampling-Based Planning. In: Brafman, R.I., Geffner, H., Hoffmann, J., Kautz, H.A. (eds.) 20th International Conference on Automated Planning and Scheduling, ICAPS 2010. pp. 242–245. AAAI (2010)
20. Ramanujan, R., Sabharwal, A., Selman, B.: Understanding Sampling Style Adversarial Search Methods. In: Grünwald, P., Spirtes, P. (eds.) 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010. pp. 474–483 (2010)
21. Ramanujan, R., Sabharwal, A., Selman, B.: On the Behavior of UCT in Synthetic Search Spaces. In: ICAPS 2011 Workshop on Monte-Carlo Tree Search: Theory and Applications (2011)
22. Ramanujan, R., Selman, B.: Trade-Offs in Sampling-Based Adversarial Planning. In: Bacchus, F., Domshlak, C., Edelkamp, S., Helmert, M. (eds.) 21st International Conference on Automated Planning and Scheduling, ICAPS 2011. AAAI (2011)
23. Sato, Y., Takahashi, D., Grimbergen, R.: A Shogi Program Based on Monte-Carlo Tree Search. ICGA Journal 33(2), 80–92 (2010)
24. Silver, D., Tesauro, G.: Monte-Carlo Simulation Balancing. In: Danyluk, A.P., Bottou, L., Littman, M.L. (eds.) 26th Annual International Conference on Machine Learning, ICML 2009. ACM International Conference Proceeding Series, vol. 382, pp. 945–952. ACM (2009)
25. Sturtevant, N.R.: An Analysis of UCT in Multi-Player Games. ICGA Journal 31(4), 195–208 (2008)
26. Winands, M.H.M., Björnsson, Y., Saito, J.T.: Monte Carlo Tree Search in Lines of Action. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 239–250 (2010)
27. Winands, M.H.M., Björnsson, Y.: Alpha-Beta-based Play-outs in Monte-Carlo Tree Search. In: Cho, S.B., Lucas, S.M., Hingston, P. (eds.) 2011 IEEE Conference on Computational Intelligence and Games, CIG 2011. pp. 110–117. IEEE (2011)
28. Winands, M.H.M., Björnsson, Y., Saito, J.T.: Monte-Carlo Tree Search Solver. In: van den Herik et al. [11], pp. 25–36