MCTS-Minimax Hybrids

Hendrik Baier and Mark H. M. Winands, *Member, IEEE*

***Abstract*—*Monte-Carlo Tree Search* (MCTS) is a sampling-based search algorithm that is state of the art in a variety of games. In many domains, its Monte-Carlo rollouts of entire games give it a strategic advantage over traditional depth-limited minimax search with $\alpha\beta$ pruning. These rollouts can often detect long-term consequences of moves, freeing the programmer from having to capture these consequences in a heuristic evaluation function. But due to its highly selective tree, MCTS runs a higher risk than full-width minimax search of missing individual moves and falling into traps in tactical situations.**

This article proposes *MCTS-minimax hybrids* that integrate shallow minimax searches into the MCTS framework. Three approaches are outlined, using minimax in the selection/expansion phase, the rollout phase, and the backpropagation phase of MCTS. Without assuming domain knowledge in the form of evaluation functions, these hybrid algorithms are a first step towards combining the strategic strength of MCTS and the tactical strength of minimax. We investigate their effectiveness in the test domains of Connect-4, Breakthrough, Othello, and Catch the Lion, and relate this performance to the tacticality of the domains.

I. Introduction

MONTE-CARLO TREE SEARCH (MCTS) [1], [2] is a best-first tree search algorithm based on simulated games as state evaluations. It samples moves instead of considering all legal moves from a given state, which allows it to handle large search spaces with high branching factors. It also uses Monte-Carlo simulations that make it independent of a static heuristic evaluation function to compare non-terminal states.

While MCTS has shown considerable success in a variety of game domains, there are still a number of games such as Chess and Checkers in which the traditional approach to adversarial planning, minimax search with $\alpha\beta$ pruning [3], remains superior. This weakness of MCTS cannot always be explained by the existence of effective evaluation functions for these games, as evaluation functions have been successfully combined with MCTS to produce strong players in games such as Amazons and Lines of Action [4], [5].

Since MCTS builds a highly selective search tree, focusing only on the most promising lines of play, it has been conjectured that it could be less appropriate than traditional, non-selective minimax search in domains containing a large number of terminal states and *shallow traps* [6]. In trap situations such as those frequent in Chess, precise tactical play is required to avoid immediate loss. MCTS methods based on sampling could easily miss a crucial move or underestimate the significance of an encountered terminal state due to averaging value backups. Conversely, MCTS could be more effective in domains such as Go, where terminal states and potential traps do not occur until the latest stage of the game. MCTS can here fully play out its strategic and positional understanding resulting from Monte-Carlo simulations of entire games.

This article explores ways of *combining* the strategic strength of MCTS and the tactical strength of minimax in order to produce more universally useful hybrid search algorithms. We do not assume the existence of heuristic evaluation functions, allowing the MCTS-minimax hybrids to be applied in any domain where MCTS is used without such heuristics (e.g. General Game Playing). The three proposed approaches use minimax search in the selection/expansion phase, the rollout phase, and the backpropagation phase of MCTS. We investigate their effectiveness in the test domains of Connect-4, Breakthrough, Othello, and Catch the Lion.

This article extends on [7]. It includes experiments in two additional test domains, which also allows for further analysis of the results. The tacticality of the domains is quantified by measuring the density and difficulty of shallow traps, and the performance of the MCTS-minimax hybrids is related to these measures.

The article is structured as follows. Section II provides background on MCTS-Solver as the baseline algorithm. Section III gives a brief overview of related work on the relative strengths of minimax and MCTS, as well as attempts at combining or nesting tree search algorithms. Section IV describes three ways of incorporating shallow-depth minimax searches into the different parts of the MCTS framework, and Section V shows experimental results of these MCTS-minimax hybrids in the four test domains. Conclusions and future research follow in Section VI.

II. Background

MCTS is the underlying framework of the algorithms in this article. It works by repeating the following four-phase loop until computation time runs out [8]. The root node 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. The selection policy should balance the exploitation of states with high value estimates and the exploration of states with uncertain value estimates. In this article, the popular UCT variant of MCTS is used, with the UCB1 policy as selection policy [9].

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 article, 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

The authors are with the Games and AI Group, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands.
e-mail: {hendrik.baier,m.winands}@maastrichtuniversity.nl

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 [10]. In this article, we are using MCTS with the *MCTS-Solver* extension [11] 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.

III. Related Work

The research of Ramanujan et al. [6], [12], [13] has repeatedly dealt with characterizing search space properties that influence the performance of MCTS relative to minimax search. *Shallow traps* were identified in [6] as a feature of domains that are problematic for MCTS, in particular Chess. Informally, the authors define a *level-k search trap* as the possibility of a player to choose an unfortunate move such that *after* executing the move, the opponent has a guaranteed winning strategy at most k plies deep. While such traps at shallow depths of 3 to 7 are not found in Go until the latest part of the endgame, they are relatively frequent in Chess games even at grandmaster level [6], partly explaining the problems of MCTS in this domain. A resulting hypothesis is that in regions of a search space containing no or very few terminal positions, shallow traps should be rare and MCTS variants should make comparatively better decisions, which was confirmed in [12] for the game of Kalah (called Mancala by the authors). In [13] finally, an artificial game tree model was used to explore the dependence of MCTS performance on the density of traps in the search space. A similar problem to shallow traps was presented in [14] under the name of *optimistic moves*—seemingly strong moves that can be refuted right away by the opponent, but take MCTS prohibitively many simulations to find the refutation. One of the motivations of the work in this article was to employ shallow-depth minimax searches within MCTS to increase the visibility of shallow traps and allow MCTS to avoid them more effectively.

In the context of General Game Playing, [15] compared the performance of minimax with $\alpha\beta$ pruning and MCTS. Restricted to the class of turn-taking, two-player, zero-sum games we are addressing here, the author identified a stable and accurate evaluation function as well as a relatively low branching factor as advantages for minimax over MCTS. In this article, we explore the use of minimax within the MCTS framework even when no evaluation function is available.

One method of combining different tree search algorithms that was proposed in the literature is the use of shallow minimax searches in every step of the MCTS *rollout phase*. This was typically restricted to checking for decisive and anti-decisive moves, as in [16] and [17] for the game of Havannah. 2-ply searches have been applied to the rollout phase in Lines of Action [18], Chess [19], as well as various multi-player games [20]. However, the existence of a heuristic evaluation function was assumed here. For MCTS-Solver, a 1-ply lookahead for winning moves in the *selection phase* at leaf nodes has already been proposed in [11], but was not independently evaluated. A different hybrid algorithm $UCTMAX_H$ was proposed in [12], employing minimax backups in an MCTS framework. However, again a strong heuristic evaluator was assumed as a prerequisite. In our work, we explore the use of minimax searches of various depths without any domain knowledge beyond the recognition of terminal states. Minimax in the rollout phase is covered in Subsection IV-A.

Furthermore, the idea of nesting search algorithms has been used in [21] and [22] to create Nested Monte-Carlo Search and Nested Monte-Carlo Tree Search, respectively. In this article, we are not using search algorithms recursively, but nesting two different algorithms in order to combine their strengths: MCTS and minimax.

IV. Hybrid Algorithms

In this section, we describe three different approaches for applying minimax with $\alpha\beta$ pruning within the MCTS framework.

A. Minimax in the Rollout Phase

While uniformly random move choices in the rollout are sufficient to guarantee the convergence of MCTS to the optimal policy, more informed rollout strategies typically greatly improve performance [23]. For this reason, it seems natural to use fixed-depth minimax searches for choosing rollout moves. Since we do not use evaluation functions in this article, minimax can only find forced wins and avoid forced losses, if possible, within its search horizon. If minimax does not find a win or loss, we return a random move. The algorithm is illustrated in Figure 1.

This strategy thus improves the quality of play in the rollouts by avoiding certain types of blunders. It informs tree growth by providing more accurate rollout returns. We call this strategy *MCTS-MR* for *MCTS with Minimax Rollouts.*

B. Minimax in the Selection and Expansion Phases

Minimax searches can also be embedded in the phases of MCTS that are concerned with traversing the tree from root to leaf: the selection and expansion phases. This strategy can use a variety of possible criteria to choose whether or not to trigger a minimax search at any state encountered during the traversal. In the work described in this article, we experimented with starting a minimax search as soon as a state has reached a given number of visits (for 0 visits, this would include the expansion phase). Figure 2 illustrates the process. Other possible criteria include e.g. starting a minimax search for a loss as soon as a given number of moves from

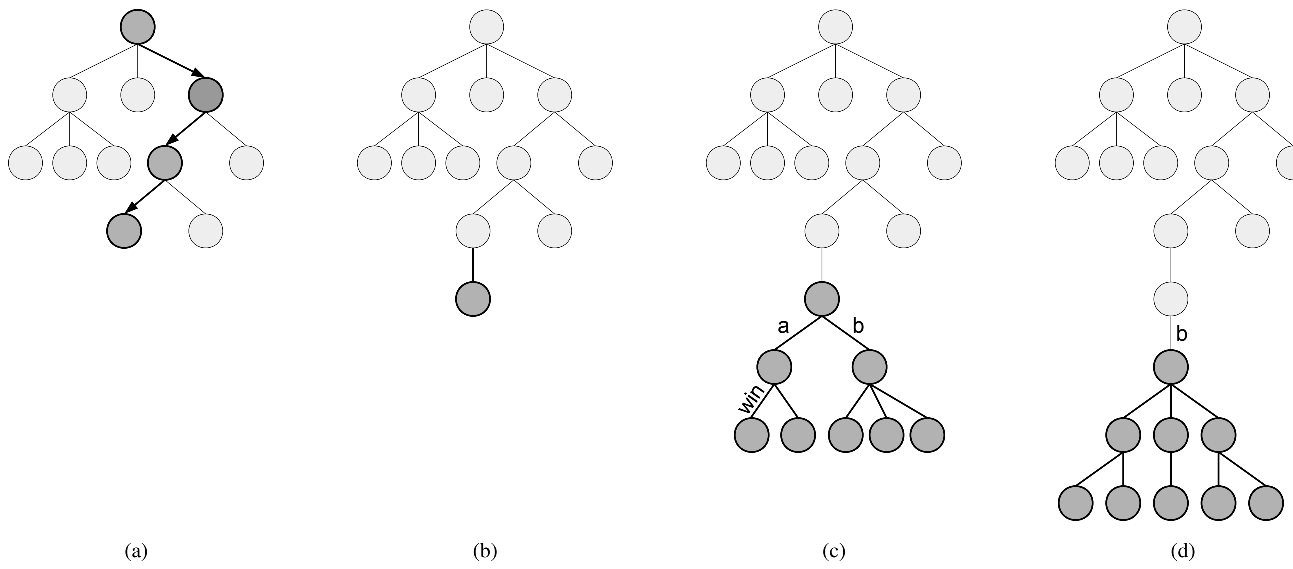


Fig. 1. The MCTS-MR hybrid. (a) The selection phase. (b) The expansion phase. (c) A minimax search is started to find the first rollout move. Since the opponent has a winning answer to move a, move b is chosen instead in this example. (d) Another minimax search is conducted for the second rollout move. In this case, no terminal states are found and a random move choice will be made.

a state have already been proven to be losses, or starting a minimax search for a loss as soon as average returns from a node fall below a given threshold (or searching for a win as soon as returns exceed a given threshold, conversely), or starting a minimax search whenever average rollout lengths from a node are short, suggesting proximity of terminal states. According to preliminary experiments, the simple criterion of visit count seemed most promising, which is why it was used in the remainder of this article. Furthermore, we start independent minimax searches for each legal move from the node in question, which allows to store proven losses for individual moves even if the node itself cannot be proven to be a loss.

This strategy improves MCTS search by performing shallow-depth, full-width checks of the immediate descendants of a subset of tree nodes. It guides tree growth by avoiding shallow losses, as well as detecting shallow wins, within or close to the MCTS tree. We call this strategy *MCTS-MS* for *MCTS with Minimax Selection*.

C. Minimax in the Backpropagation Phase

As mentioned in Subsection II, MCTS-Solver tries to propagate game-theoretic values (*proven win* and *proven loss*) as far up the tree as possible, starting from the terminal state visited in the current simulation. It has to switch to regular rollout returns (*win* and *loss*) as soon as at least one sibling of a proven loss move is not marked as proven loss itself. Therefore, we employ shallow minimax searches whenever this happens, actively searching for proven losses instead of hoping for MCTS-Solver to find them in future simulations. If minimax succeeds at proving all moves from a given state s to be losses, we can backpropagate a *proven loss* instead of just a *loss* to the next-highest tree level—i.e. a proven win for the opponent player's move leading to s (see a negamax formulation of this algorithm in Figure 3).

This strategy improves MCTS-Solver by providing the backpropagation step with helpful information whenever possible, which allows for quicker proving and exclusion of moves from further MCTS sampling. Other than the strategies described in IV-A and IV-B, it only triggers when a terminal position has been found in the tree and the MCTS-Solver extension applies. For this reason, it avoids spending computation time on minimax searches in regions of the search space with no or very few terminal positions. Minimax can also search deeper each time it is triggered, because it is triggered less often. We call this strategy *MCTS-MB* for *MCTS with Minimax Backpropagation*.

V. Experimental Results

We tested the MCTS-minimax hybrids in four different domains: The two-player, zero-sum games of *Connect-4*, *Breakthrough*, *Othello*, and *Catch the Lion*. In all experimental conditions, we compared the hybrids against regular MCTS-Solver as the baseline. UCB1-TUNED [9] is used as selection policy. The exploration factor C of UCB1-TUNED was optimized once for MCTS-Solver in all games and then kept constant for both MCTS-Solver and the MCTS-minimax hybrids during testing. Optimal values were 1.3 in Connect-4, 0.8 in Breakthrough, 0.7 in Othello, and 0.7 in Catch the Lion. Draws, which are possible in Connect-4 and Othello, were counted as half a win for both players. We used minimax with $\alpha\beta$ pruning, but no other search enhancements. Unless stated otherwise, computation time was 1 second per move.

Note that Figures 11 to 22 show the results of parameter tuning experiments. The best-performing parameter values found during tuning were tested with an *additional* 5000 games after each tuning experiment. The results of these replications are reported in the text.

To ensure a minimal standard of play, the MCTS-Solver

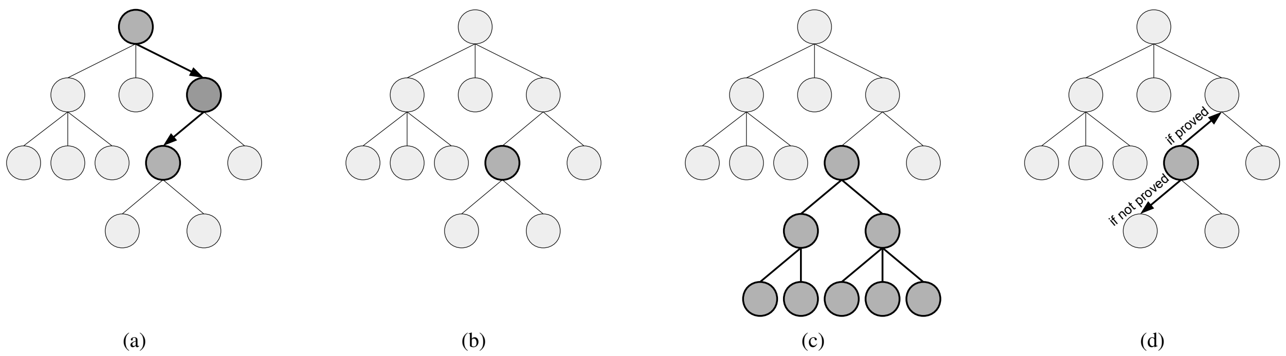


Fig. 2. The MCTS-MS hybrid. (a) Selection and expansion phases. The tree is traversed in the usual fashion until a node satisfying the minimax trigger criterion is found. (b) In this case, the marked node has reached a prespecified number of visits. (c) A minimax search is started from the node in question. (d) If the minimax search has proved the node's value, this value can be backpropagated. Otherwise, the selection phase continues as normal.

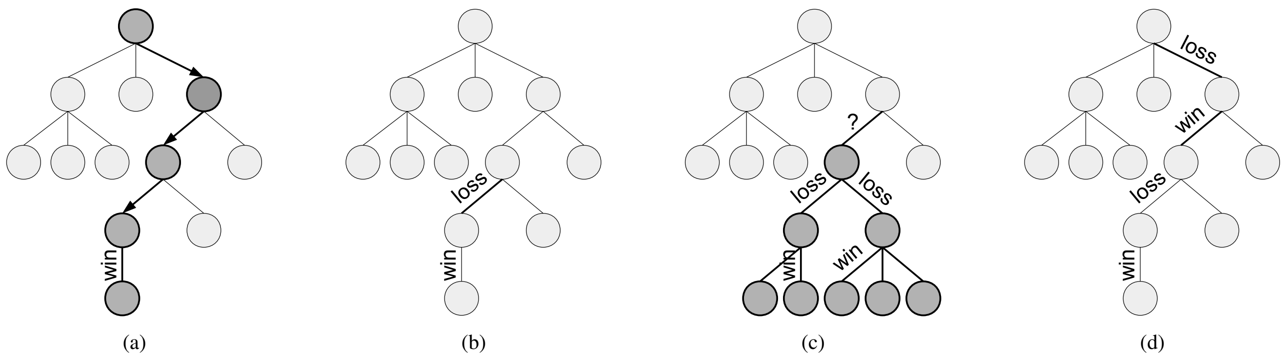


Fig. 3. The MCTS-MB hybrid. (a) Selection and expansion phases. The expanded move wins the game. (b) This implies the opponent's previous move was proven to be a loss. (c) A minimax search is triggered in order to check whether the move marked by "?" can be proven to be a win. In this example, all opponent answers are proven losses, so it can. (d) This also implies the opponent's previous move was proven to be a loss. The root state's value is now proven.

baseline was tested against a random player. MCTS-Solver won 100% of 1000 games in all four domains.

A. Games

This section outlines the rules of the four test domains.

1) Connect-4: Connect-4 is played on a 7×6 board. At the start of the game, the board is empty. The two players alternatingly place white and black discs in one of the seven columns, always filling the lowest available space of the chosen column. Columns with six discs are full and cannot be played anymore. The game is won by the player who succeeds first at connecting four tokens of her own color either vertically, horizontally, or diagonally. If the board is filled completely without any player reaching this goal, the game ends in a draw.

2) Breakthrough: The variant of Breakthrough used in this article 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 the opponent, i.e. reaching the first row as Black or reaching the last row as White.

3) Othello: 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.

4) Catch the Lion: Catch the Lion is a simplified form of Shogi (see [24] 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 [6].

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 the 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.

B. Density and Difficulty of Shallow Traps

In order to measure an effect of employing shallow minimax searches without an evaluation function within MCTS, terminal states have to be present in sufficient density throughout the search space, in particular the part of the search space relevant at our level of play. We played 1000 self-play games of MCTS-Solver in all domains to test this property, using 1 second per move. At each turn, we determined whether there exists *at least one* trap at depth (up to) 3 for the player to move. The same methodology was used in [6].

Figures 4, 5, 6, and 7 show that shallow traps are indeed found throughout most domains, which suggests improving the ability of MCTS to identify and avoid such traps is worthwhile. Traps appear most frequently in Catch the Lion—a highly tactical domain—followed by Breakthrough and Connect-4. A game of Othello usually only ends when the board is completely filled however, which explains why traps only appear when the game is nearly over. Furthermore, we note that in contrast to Breakthrough and Othello the density of traps for both players differs significantly in Connect-4 and in the early phase of Catch the Lion. Finally, we see that Breakthrough games longer than 40 turns, Othello games longer than 60 turns and Catch the Lion games longer than 50–60 moves are rare, which explains why the data become more noisy.

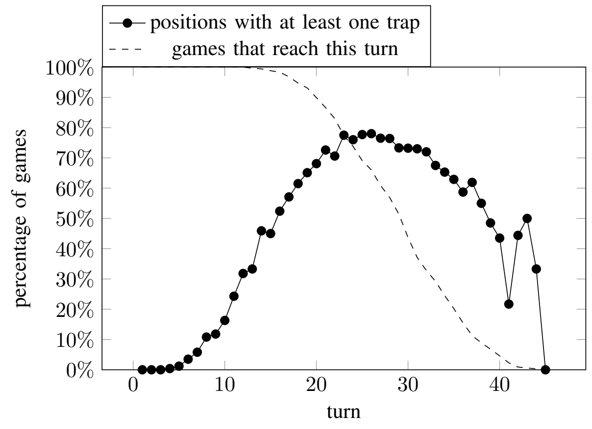


Fig. 5. Density of level-3 search traps in Breakthrough.

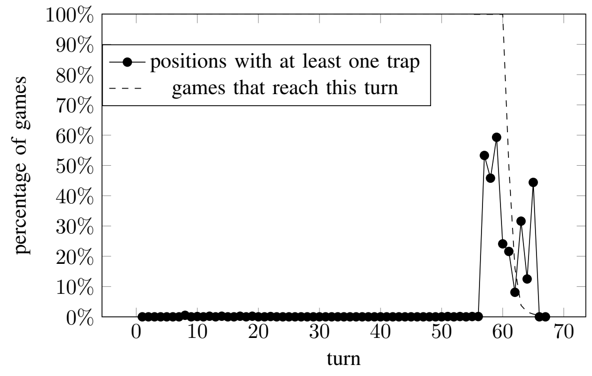


Fig. 6. Density of level-3 search traps in Othello.

In order to provide a more condensed view of the data, Figure 8 compares the *average number* of level-3 to level-7 search traps over all positions encountered in the test games. These were 34187 positions in Catch the Lion, 28344 positions in Breakthrough, 36633 positions in Connect-4, and 60723 positions in Othello. Note that a level-k trap requires a winning strategy *at most* k plies deep, which means every level-k trap is a level-$(k+1)$ trap as well. As Figure 8 shows, Catch the Lion is again leading in trap density, followed by

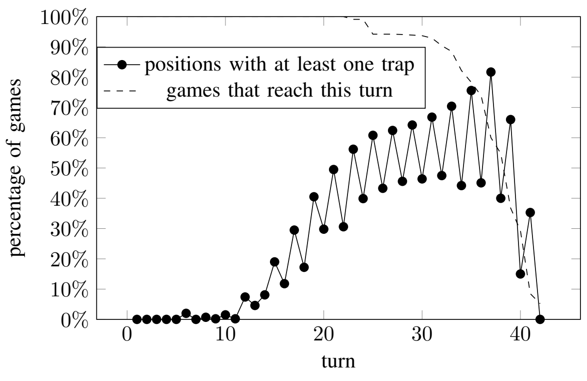


Fig. 4. Density of level-3 search traps in Connect-4.

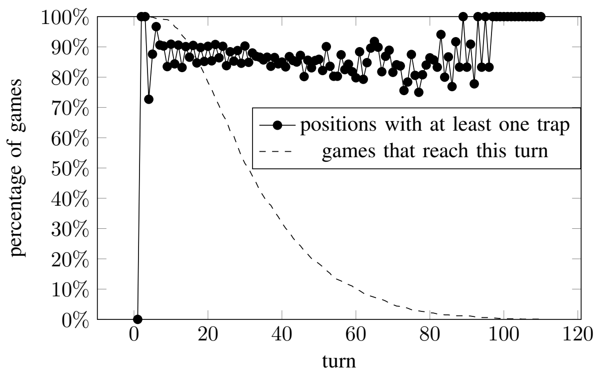


Fig. 7. Density of level-3 search traps in Catch the Lion.

Breakthrough, Connect-4, and finally Othello with a negligible average number of traps.

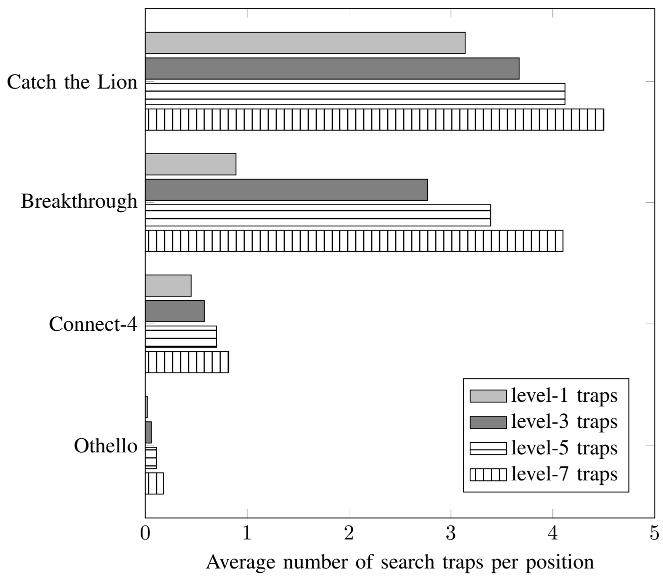


Fig. 8. Comparison of trap density in Catch the Lion, Breakthrough, Connect-4, and Othello.

In an additional experiment, we tried to quantify the average *difficulty* of traps for MCTS. MCTS is more likely to "fall into" a trap, i.e. waste much effort on exploring a trap move, if rollouts starting with this move frequently return a misleading winning result instead of the correct losing result. Depending on the search space, trap moves might be relatively frequent but still easy to avoid for MCTS because they get resolved correctly in the rollouts—or less frequent but more problematic due to systematic errors in rollout returns. Therefore, 1000 random rollouts were played starting with every level-3 to level-7 trap move found in the test games. No tree was built during sampling. Any rollout return other than a loss was counted as incorrect. Figure 9 shows the proportion of rollouts that returned the incorrect result, averaged over all traps.

We observe that in the set of four test domains, the domains with the highest average *number* of traps are also the domains with the highest expected *difficulty* of traps. Catch the Lion is again followed by Breakthrough, Connect-4, and Othello in last place with near-perfect random evaluation of traps. This property of Othello can be explained with the fact that almost all traps appear at the end of the game when the board is filled, and the last filling move has no alternatives. Thus, the opponent can make one mistake less than in other games when executing the trap, and in many situations executing the trap is the only legal path.

In conclusion, due to both the difference in trap density as well as trap difficulty, we expect MCTS-minimax hybrids to work relatively better in domains like Catch the Lion than in domains like Othello. In Subsection V-H, the observations of this section are related to the performance of the MCTS-minimax hybrids presented in the following, and this expectation is confirmed.

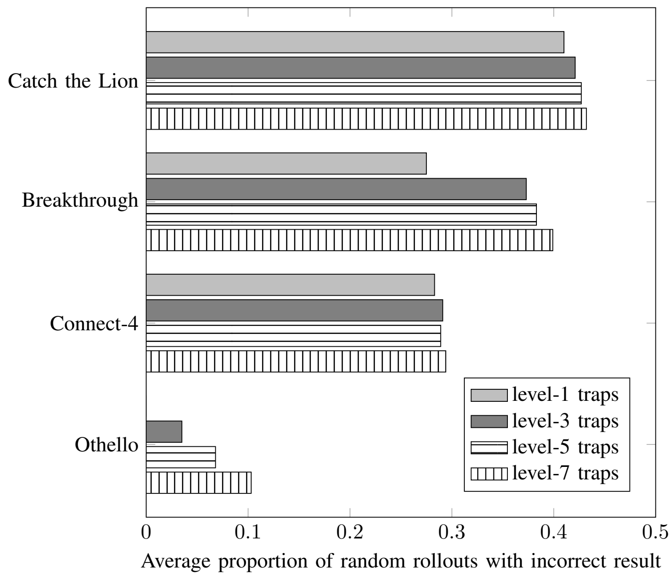


Fig. 9. Comparison of trap difficulty in Catch the Lion, Breakthrough, Connect-4, and Othello.

C. Connect-4

In this subsection, we summarize the experimental results in the game of Connect-4. The baseline MCTS-Solver implementation performs about 91000 simulations per second when averaged over an entire game.

1) Minimax in the Rollout Phase: We tested minimax at search depths 1 ply to 4 plies in the rollout phase of a Connect-4 MCTS-Solver player. Each resulting player, abbreviated as MCTS-MR-1 to MCTS-MR-4, played 1000 games against regular MCTS-Solver with uniformly random rollouts. Figure 11 presents the results.

Minimax is computationally more costly than a random rollout policy. MCTS-MR-1 finishes about 69% as many simulations per second as the baseline, MCTS-MR-2 about 31% as many, MCTS-MR-3 about 11% as many, MCTS-MR-4 about 7% as many when averaged over one game of Connect-4. This typical speed-knowledge trade-off explains the decreasing performance of MCTS-MR for higher minimax search depths, although the quality of rollouts increases. Remarkably, MCTS-MR-1 performs significantly worse than the baseline. This also held when we performed the comparison using equal numbers of MCTS iterations (10^5) per move instead of equal time (1 second) per move for both players. In this scenario, we found MCTS-MR-1 to achieve a win rate of 36.3% in 1000 games against the baseline. We suspect this is due to some specific imbalance in Connect-4 rollouts with depth-1 minimax—it has been repeatedly found that the strength of a rollout policy as a standalone player is not always a good predictor of its strength when used within a Monte-Carlo search algorithm (see [25] and [26] for similar observations in the game of Go using naive Monte-Carlo and MCTS, respectively).

In order to illustrate this phenomenon, Figure 10 gives an intuition for situations in which depth-1 minimax rollouts can be less effective for state evaluation than uniformly random rollouts. The figure shows the partial game tree (not the search tree) of a position that is a game-theoretic win for the root player. Only if the root player does not choose move a at the root state can the opponent prevent the win. The building of a search tree is omitted for simplification in the following. If we start a uniformly random rollout from the root state, we will get the correct result with a probability of $\frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{2}{3} = 0.75$. If we use a depth-1 minimax rollout however, the root player's opponent will always be able to make the correct move choice at states A, B, and C, leading to an immediate loss for the root player. As a result, the correct result will only be found with a probability of $\frac{1}{4} \cdot 1 + \frac{3}{4} \cdot 0 = 0.25$. If similar situations appear in sufficient frequency in Connect-4, they could provide an explanation for systematic evaluation errors of MCTS-MR-1.

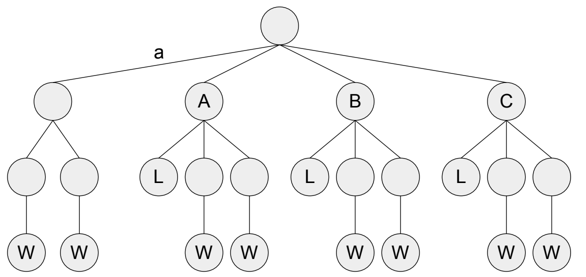


Fig. 10. A problematic situation for MCTS-MR-1 rollouts. The figure represents a partial game tree. “L” and “W” mark losing and winning states from the point of view of the root player.

In the Connect-4 experiments, MCTS-MR-2 outperformed all other variants. Over an entire game, it completed about 28000 simulations per second on average. In an additional 5000 games against the baseline, it won 72.1% (95% confidence interval: 70.8%−73.3%) of games, which is a significant improvement (p<0.001).

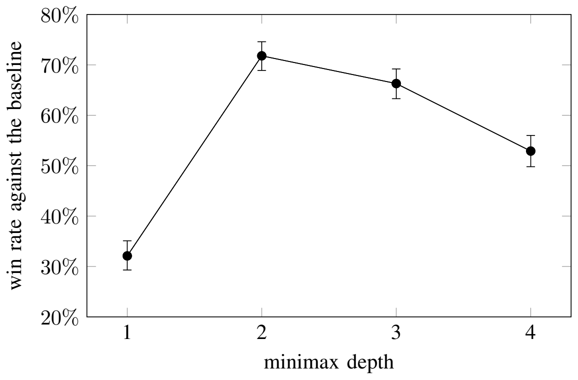


Fig. 11. Performance of MCTS-MR in Connect-4.

2) Minimax in the Selection and Expansion Phases: The variant of MCTS-MS we tested starts a minimax search from a state in the tree if that state has reached a fixed number of visits when encountered by the selection policy. We call this variant, using a minimax search of depth d when reaching v visits, *MCTS-MS-d-Visit-v*. If the visit limit is set to 0, this means every tree node is searched immediately in the expansion phase even before it is added to the tree.

We tested MCTS-MS-d-Visit-v for $d \in \{2, 4\}$ and $v \in \{0, 1, 2, 5, 10, 20, 50, 100\}$. We found it to be most effective to set the $\alpha\beta$ search window such that minimax was only used to detect forced losses (traps). Since suicide is impossible in Connect-4, we only searched for even depths. Each condition consisted of 1000 games against the baseline player. The results are shown in Figure 12. Low values of v result in too many minimax searches being triggered, which slows down MCTS. High values of v mean that the tree below the node in question has already been expanded to a certain degree, and minimax might not be able to gain much new information. Additionally, high values of v result in too few minimax searches, such that they have little effect.

MCTS-MS-2-Visit-1 was the most successful condition. It played about 83700 simulations per second on average over an entire game. There were 5000 additional games played against the baseline and a total win rate of 53.6% (95% confidence interval: 52.2%−55.0%) was achieved, which is a significantly stronger performance (p<0.001).

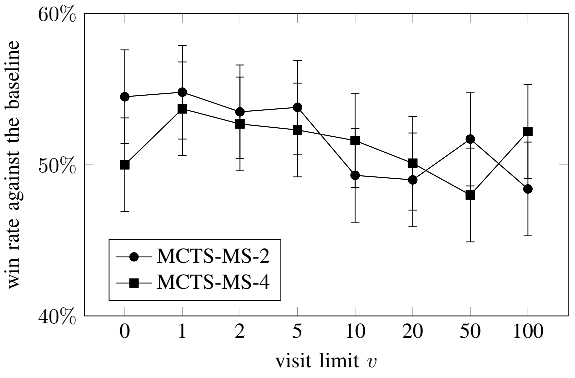


Fig. 12. Performance of MCTS-MS in Connect-4.

3) Minimax in the Backpropagation Phase: MCTS-Solver with minimax in the backpropagation phase was tested with minimax search depths 1 ply to 6 plies. Contrary to MCTS-MS as described in V-C2, we experimentally determined it to be most effective to use MCTS-MB with a full minimax search window in order to detect both wins and losses. We therefore included odd search depths. Again, all moves from a given node were searched independently in order to be able to prove their individual game-theoretic values. The resulting players were abbreviated as MCTS-MB-1 to MCTS-MB-6 and played 1000 games each against the regular MCTS-Solver baseline. The results are shown in Figure 13.

MCTS-MB-1 as the best-performing variant played 5000 additional games against the baseline and won 49.9% (95% confidence interval: 48.5% − 51.3%) of them, which shows no significant difference in performance. It played about 88500 simulations per second when averaged over the whole game.

D. Breakthrough

The experimental results in the Breakthrough domain are described in this subsection. Our baseline MCTS-Solver implementation plays about 45100 simulations per second on average.

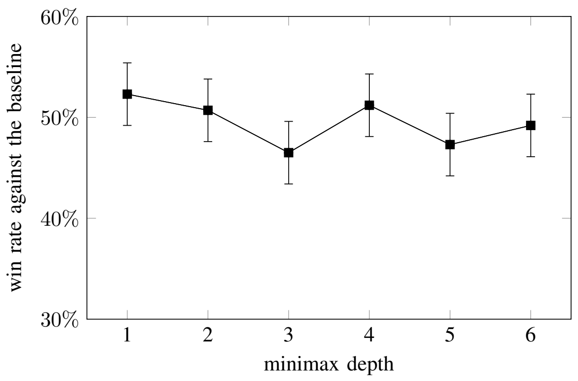


Fig. 13. Performance of MCTS-MB in Connect-4.

1) Minimax in the Rollout Phase: As in Connect-4, we tested 1-ply to 4-ply minimax searches in the rollout phase of a Breakthrough MCTS-Solver player. The resulting players MCTS-MR-1 to MCTS-MR-4 played 1000 games each against regular MCTS-Solver with uniformly random rollouts. The results are presented in Figure 14.

Interestingly, all MCTS-MR players were significantly weaker than the baseline ($p<0.001$). The advantage of a 1- to 4-ply lookahead in rollouts does not seem to outweigh the computational cost in Breakthrough, possibly due to the larger branching factor, longer rollouts, and more time-consuming move generation than in Connect-4. MCTS-MR-1 searches only about 15.8% as fast as the baseline, MCTS-MR-2 about 2.3% as fast, MCTS-MR-3 about 0.5% as fast, MCTS-MR-4 about 0.15% as fast when measured in simulations completed in a one-second search of the empty Connect-4 board. When comparing with equal numbers of MCTS iterations (10000) per move instead of equal time (1 second) per move for both players, MCTS-MR-1 achieved a win rate of 67.6% in 1000 games against the baseline. MCTS-MR-2 won 83.2% of 1000 games under the same conditions. It may be possible to optimize our Breakthrough implementation. However, as the following subsections indicate, application of minimax in other phases of MCTS seems to be the more promising approach in this game.

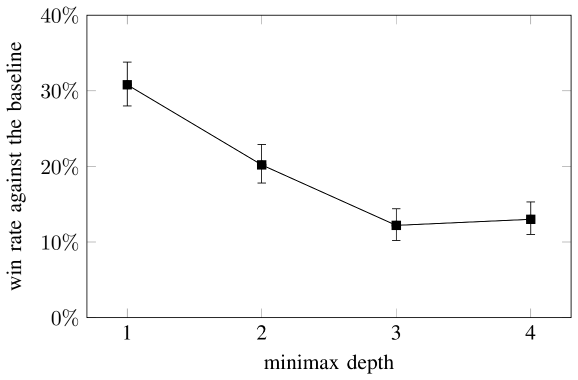


Fig. 14. Performance of MCTS-MR in Breakthrough.

2) Minimax in the Selection and Expansion Phases: We tested the same variants of MCTS-MS for Breakthrough as for Connect-4: MCTS-MS-d-Visit-v for $d \in \{2,4\}$ and $v \in \{0, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\}$. 1000 games against the baseline player were played for each experimental condition. Figure 15 shows the results.

MCTS-MS-2-Visit-2 appeared to be the most effective variant. When averaged over the whole game, it performed about 33000 simulations per second. 5000 additional games against the baseline confirmed a significant increase in strength ($p<0.001$) with a win rate of 67.3% (95% confidence interval: $66.0\% - 68.6\%$).

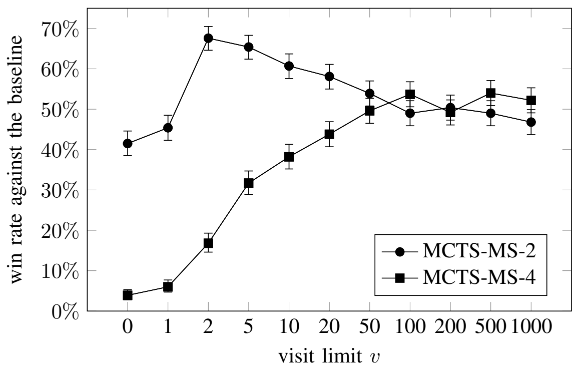


Fig. 15. Performance of MCTS-MS in Breakthrough.

3) Minimax in the Backpropagation Phase: MCTS-MB-1 to MCTS-MB-6 were tested analogously to Connect-4, playing 1000 games each against the regular MCTS-Solver baseline. Figure 16 presents the results.

The best-performing setting MCTS-MB-2 played 5000 additional games against the baseline and won 60.6% (95% confidence interval: $59.2\% - 62.0\%$) of them, which shows a significant improvement ($p<0.001$). It played about 46800 simulations per second on average.

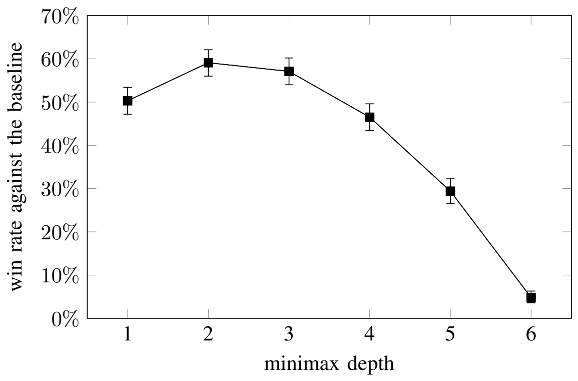


Fig. 16. Performance of MCTS-MB in Breakthrough.

E. Othello

This subsection describes the experimental results in Othello. Our baseline MCTS-Solver implementation plays about 8700 simulations per second on average in this domain.

1) Minimax in the Rollout Phase: Figure 17 presents the results of MCTS-MR-1 to MCTS-MR-4 playing 1000 games each against the MCTS-Solver baseline. None of the MCTS-MR conditions tested had a positive effect on playing strength. The best-performing setting MCTS-MR-1 played 5000 additional games against the baseline and won 43.7% (95% confidence interval: $42.3\% - 45.1\%$) of them, which is significantly weaker than the baseline (p<0.001). MCTS-MR-1 simulated about 6400 games per second on average.

When playing with equal numbers of MCTS iterations per move, MCTS-MR-1 won 47.9% of 1000 games against the baseline (at 5000 rollouts per move), MCTS-MR-2 won 54.3% (at 1000 rollouts per move), MCTS-MR-3 won 58.2% (at 250 rollouts per move), and MCTS-MR-4 won 59.6% (at 100 rollouts per move). This shows that there is a positive effect—ignoring the time overhead—of minimax searches in the rollouts, even though these searches can only return useful information in the very last moves of an Othello game. It could be worthwhile in Othello and similar games to restrict MCTS-MR to minimax searches only in these last moves.

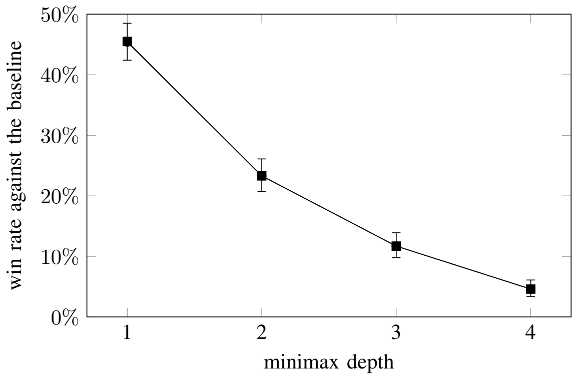


Fig. 17. Performance of MCTS-MR in Othello.

2) Minimax in the Selection and Expansion Phases: Again, MCTS-MS-d-Visit-v was tested for $d \in \{2, 4\}$ and $v \in \{$0, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000$\}$. Each condition played 1000 games against the baseline player. Figure 18 presents the results.

The best-performing version, MCTS-MS-2-Visits-50, won 50.8% (95% confidence interval: $49.4\% - 52.2\%$) of 5000 additional games against the baseline. Thus, no significant difference in performance was found. The speed was about 8200 rollouts per second.

3) Minimax in the Backpropagation Phase: MCTS-MB-1 to MCTS-MB-6 played 1000 games each against the regular MCTS-Solver baseline. The results are shown in Figure 19.

MCTS-MB-2, the most promising setting, achieved a result of 49.2% (95% confidence interval: $47.8\% - 50.6\%$) in 5000 additional games against the baseline. No significant performance difference to the baseline could be shown. The hybrid played about 8600 rollouts per second on average.

In conclusion, no effect of the three tested MCTS-minimax hybrids could be shown in Othello, the domain with the lowest number of traps examined in this article.

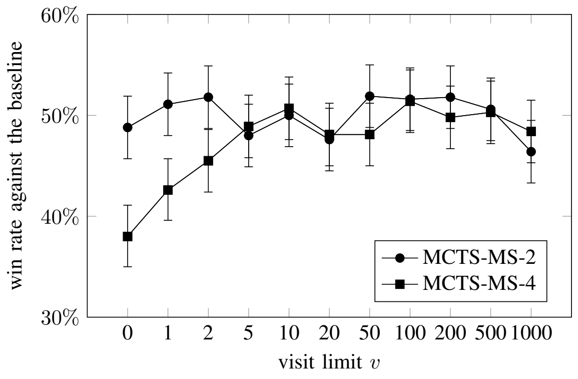


Fig. 18. Performance of MCTS-MS in Othello.

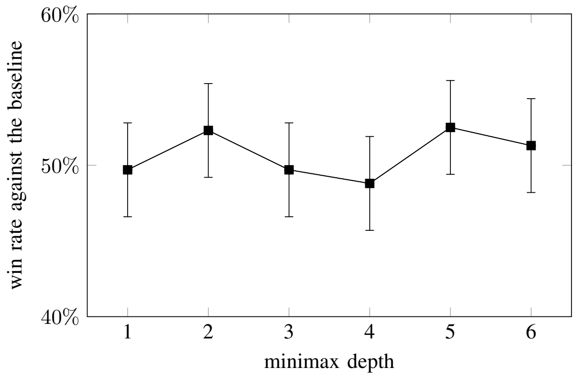


Fig. 19. Performance of MCTS-MB in Othello.

F. Catch the Lion

In this subsection, the results of testing in the domain of Catch the Lion are presented. In this game, the baseline MCTS-Solver plays approximately 34700 simulations per second on average.

1) Minimax in the Rollout Phase: Figure 20 presents the results of MCTS-MR-1 to MCTS-MR-4 playing 1000 games each against the MCTS-Solver baseline.

There is an interesting even-odd effect, with MCTS-MR seemingly playing stronger at odd minimax search depths. Catch the Lion is the only domain of the four where MCTS-MR-3 was found to perform better than MCTS-MR-2. MCTS-MR-1 played best in these initial experiments and was tested in an additional 5000 games against the baseline. The hybrid won 94.8% (95% confidence interval: $94.1\% - 95.4\%$) of these, which is significantly stronger (p<0.001). It reached about 28700 rollouts per second.

2) Minimax in the Selection and Expansion Phases: MCTS-MS-d-Visit-v was tested for $d \in \{2, 4, 6\}$ and $v \in \{$0, 1, 2, 5, 10, 20, 50, 100$\}$. With each parameter setting, 1000 games were played against the baseline player. Figure 21 shows the results.

MCTS-MS-4-Visits-2 performed best of all tested settings. Of an additional 5000 games against the baseline, it won 76.8% (95% confidence interval: $75.6\% - 78.0\%$). This is a significant improvement (p<0.001). It played about 14500

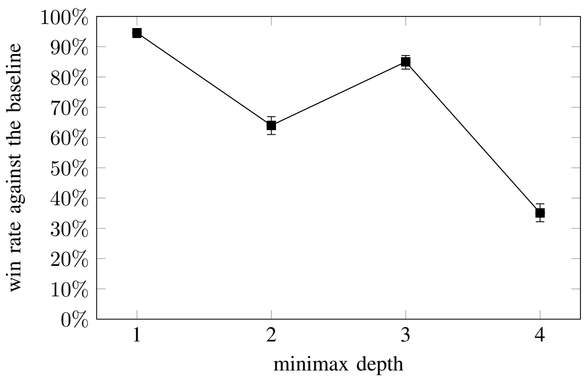


Fig. 20. Performance of MCTS-MR in Catch the Lion.

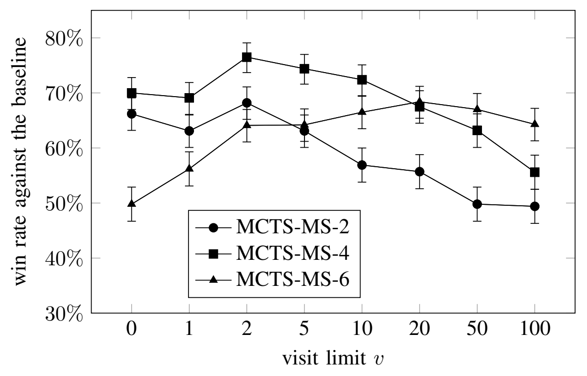


Fig. 21. Performance of MCTS-MS in Catch the Lion.

games per second on average.

3) Minimax in the Backpropagation Phase: MCTS-MB-1 to MCTS-MB-6 were tested against the baseline in 1000 games per condition. Figure 22 presents the results.

MCTS-MB-4 performed best and played 5000 more games against the baseline. It won 73.1% (95% confidence interval: $71.8\% - 74.3\%$) of them, which is a significant improvement (p<0.001). The speed was about 20000 rollouts per second.

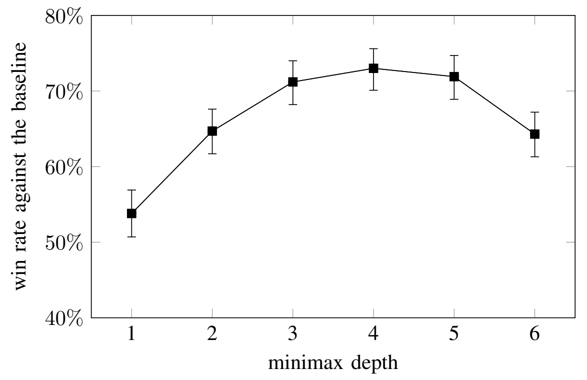


Fig. 22. Performance of MCTS-MB in Catch the Lion.

In conclusion, the domain with the highest number of traps examined in this article, Catch the Lion, also showed the strongest performance of all three MCTS-minimax hybrids.

G. Comparison of Algorithms

Sections V-C and V-D showed the performance of MCTS-MR, MCTS-MS, and MCTS-MB against the baseline player in both Connect-4 and Breakthrough. In order to facilitate comparison, we also tested the best-performing variants of these MCTS-minimax hybrids against each other. In Connect-4, MCTS-MR-2, MCTS-MS-2-Visit-1, and MCTS-MB-1 played in each possible pairing; in Breakthrough, MCTS-MR-1, MCTS-MS-2-Visit-2, and MCTS-MB-2 were chosen; in Othello, MCTS-MR-1, MCTS-MS-2-Visit-50, and MCTS-MB-2; and in Catch the Lion, MCTS-MR-1, MCTS-MS-4-Visits-2, and MCTS-MB-4. 2000 games were played in each condition. Figure 23 presents the results.

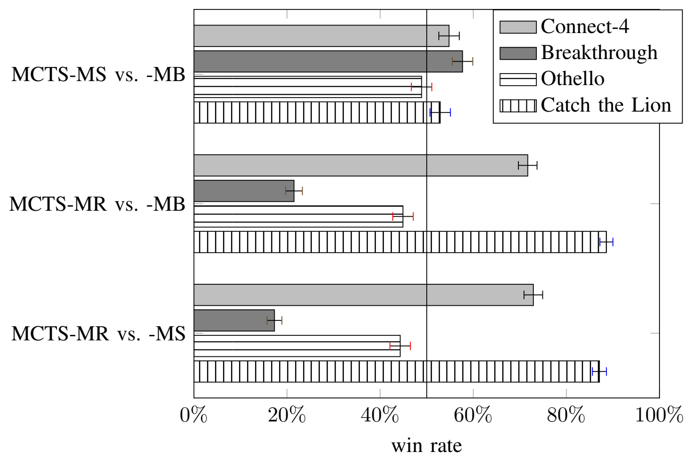


Fig. 23. Performance of MCTS-MR, MCTS-MS, and MCTS-MB against each other in Connect-4, Breakthrough, Othello, and Catch the Lion.

Consistent with the results from the previous sections, MCTS-MS outperformed MCTS-MB in Breakthrough and Connect-4, while no significant difference could be shown in Othello and Catch the Lion. MCTS-MR was significantly stronger than the two other algorithms in Connect-4 and Catch the Lion, but weaker than both in Breakthrough and Othello.

In a second experiment, the best-performing MCTS-minimax hybrids played against the baseline at different time settings from 250 ms per move to 5000 ms per move. 2000 games were played in each condition. The results are shown in Figure 24 for Connect-4, Figure 25 for Breakthrough, Figure 26 for Othello, and Figure 27 for Catch the Lion.

We can observe that at least up to 5 seconds per move, additional time makes the significant performance differences between algorithms more pronounced in most domains. While in Connect-4, it is MCTS-MR that profits most from additional time, we can see the same effect for MCTS-MS and MCTS-MB in Breakthrough. The time per move does not change the ineffectiveness of hybrid search in Othello. Interestingly however, MCTS-MR does not profit from longer search times in Catch the Lion. It is possible that in this highly tactical domain, the baseline MCTS-Solver scales better due to being faster and building larger trees. The larger trees could help avoid deeper traps than the MCTS-MR rollouts can detect.

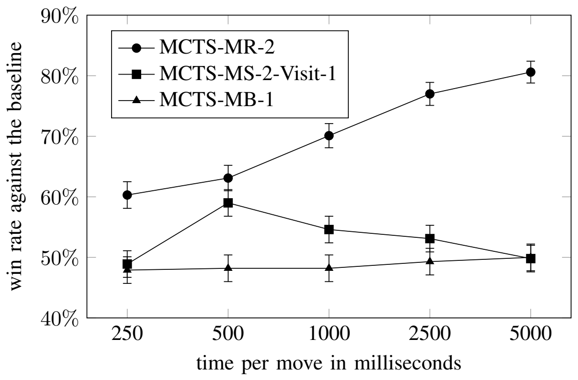


Fig. 24. Performance of MCTS-MR-2, MCTS-MS-2-Visit-1, and MCTS-MB-1 at different time settings in Connect-4.

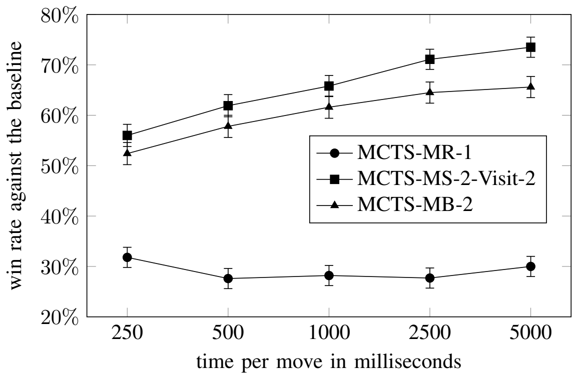


Fig. 25. Performance of MCTS-MR-1, MCTS-MS-2-Visit-2, and MCTS-MB-2 at different time settings in Breakthrough.

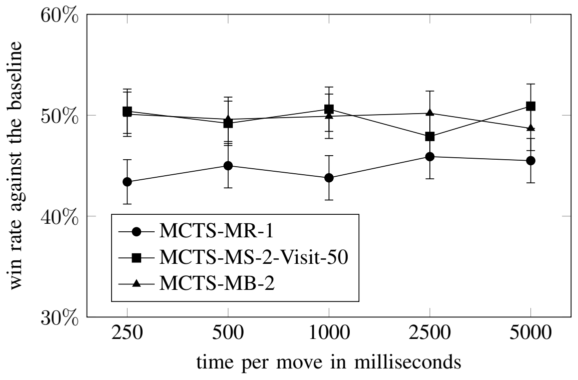


Fig. 26. Performance of MCTS-MR-1, MCTS-MS-2-Visit-50, and MCTS-MB-2 at different time settings in Othello.

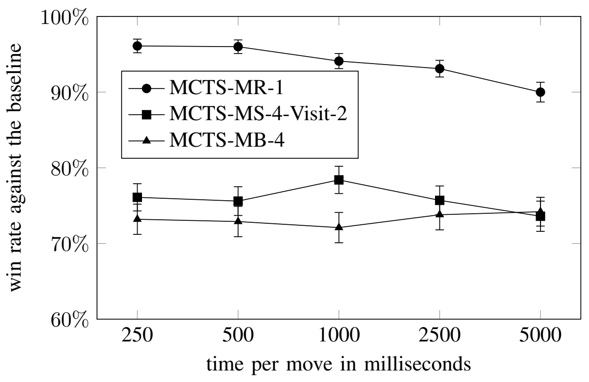


Fig. 27. Performance of MCTS-MR-1, MCTS-MS-4-Visit-2, and MCTS-MB-4 at different time settings in Catch the Lion.

H. Comparison of Domains

In Subsections V-C to V-F, the experimental results were ordered by domain. In this subsection, the data on the best-performing hybrid variants are presented again, ordered by the type of hybrid instead. Figures 28, 29, and 30 show the performance of MCTS-MS, MCTS-MB, and MCTS-MR, respectively.

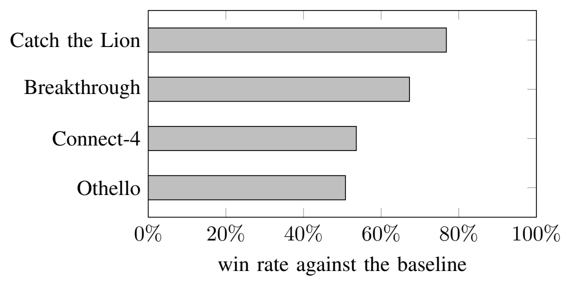


Fig. 28. Comparison of MCTS-MS performance in Catch the Lion, Breakthrough, Connect-4, and Othello. The best-performing parameter settings are compared for each domain.

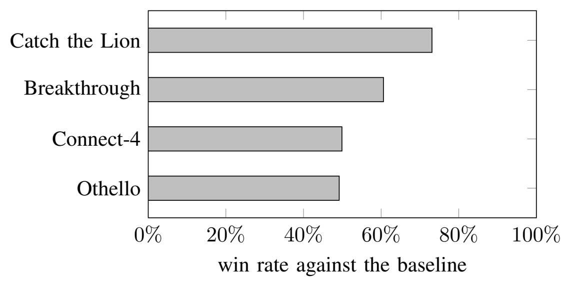


Fig. 29. Comparison of MCTS-MB performance in Catch the Lion, Breakthrough, Connect-4, and Othello. The best-performing parameter settings are compared for each domain.

Both MCTS-MS and MCTS-MB are most effective in Catch the Lion, followed by Breakthrough, Connect-4, and finally Othello, where no positive effect could be observed. The parallels to the ordering of domains with respect to trap density (Figure 8) and trap difficulty (Figure 9) are striking. As expected, these factors seem to strongly influence the relative

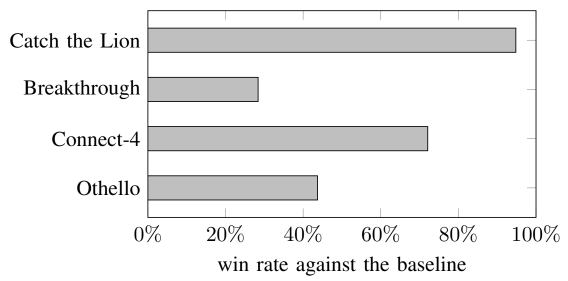


Fig. 30. Comparison of MCTS-MR performance in Catch the Lion, Breakthrough, Connect-4, and Othello. The best-performing parameter settings are compared for each domain.

effectivity of MCTS-minimax hybrids in a given domain. This order is different in MCTS-MR only due to the poor performance in Breakthrough, which may be explained by this domain having a higher average branching factor than the other three. During the selection step of MCTS, there are on average 15.5 legal moves available in Breakthrough, but only 6.4 in Connect-4, 8.1 in Othello, and 9.2 in Catch the Lion, measured in self-play games of our MCTS-Solver implementation with 1 second per move.

I. Effect of Branching Factor

In order to shed more light on the influence of the average branching factor mentioned above, the hybrids were also tested on Breakthrough boards of larger widths. In addition to the 6×6 board used in the previous experiments (average branching factor 15.5), we included the sizes 9×6 (average branching factor 24.0), 12×6 (average branching factor 37.5), 15×6 (average branching factor 43.7), and 18×6 (average branching factor 54.2) in this series of experiments. While the average game length also increases with the board width—from about 30 moves on the 6×6 board to about 70 moves on the 18×6 board—this setup served as an approximation to varying the branching factor while keeping other game properties as equal as possible (without using artificial game trees). Figure 31 presents the results, comparing the best-performing settings of MCTS-MS, MCTS-MB, and MCTS-MR across the five board sizes. The hybrids were tuned for each board size separately. Each data point represents 2000 games.

The branching factor has a strong effect on MCTS-MR, reducing the win rate of MCTS-MR-1 from 30.8% (on 6×6) to 10.2% (on 18×6). Deeper minimax searches in MCTS rollouts scale even worse: The performance of MCTS-MR-2 for example drops from 20.2% to 0.1% (not shown in the Figure). The MCTS-minimax hybrids newly proposed in this article, however, do not seem to be strongly affected by the range of branching factors examined. Both MCTS-MS and MCTS-MB were effective up to a branching factor of at least 50.

VI. Conclusion and Future Research

The strategic strength of MCTS lies to a great extent in the Monte-Carlo simulations, allowing the search to observe

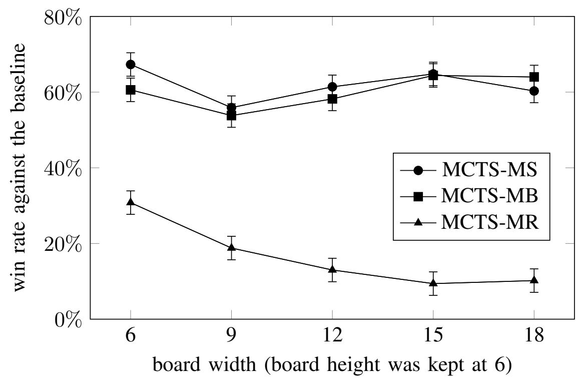


Fig. 31. Performance of MCTS-minimax hybrids across different board widths in Breakthrough.

even distant consequences of actions, if only through the observation of probabilities. The tactical strength of minimax lies largely in its exhaustive approach, guaranteeing to never miss a consequence of an action that lies within the search horizon, and backing up game-theoretic values from leaves with certainty and efficiency.

In this article, we examined three knowledge-free approaches of integrating minimax into MCTS: the application of minimax in the rollout phase with *MCTS-MR*, the selection and expansion phases with *MCTS-MS*, and the backpropagation phase with *MCTS-MB*. The newly proposed variant MCTS-MS significantly outperformed regular MCTS with the MCTS-Solver extension in Catch the Lion, Breakthrough, and Connect-4. The same holds for the proposed MCTS-MB variant in Catch the Lion and Breakthrough, while the effect in Connect-4 is neither significantly positive nor negative. The only way of integrating minimax search into MCTS known from the literature (although typically used with an evaluation function), MCTS-MR, was quite strong in Catch the Lion and Connect-4 but significantly weaker than the baseline in Breakthrough, suggesting it might be less robust with regard to differences between domains such as the average branching factor. As expected, none of the MCTS-minimax hybrids had a positive effect in Othello. The game of Go would be another domain where we do not expect any success with MCTS-minimax hybrids, because it has no trap states until the latest game phase.

With the exception of the weak performance of MCTS-MR in Breakthrough, probably mainly caused by its larger branching factor, we observed that all MCTS-minimax hybrids tend to be most effective in Catch the Lion, followed by Breakthrough, Connect-4, and finally Othello. The density and difficulty of traps, as discussed in Subsection V-B, thus seem to predict the relative performance of MCTS-minimax hybrids across domains well. In conclusion, MCTS-minimax hybrids can strongly improve the performance of MCTS in tactical domains, with MCTS-MR working best in domains with low branching factors (up to roughly 10 moves on average), and MCTS-MS and MCTS-MB being more robust against higher branching factors. This was tested for branching factors of up to roughly 50 in Breakthrough on different boards.

Preliminary experiments with combinations of the three hybrids seem to indicate that their effects are overlapping to a large degree. Combinations do not seem to perform significantly better than the best-performing individual hybrids in the domain at hand. This could still be examined in more detail.

According to our observations, problematic domains for MCTS-minimax hybrids seem to feature a low density of traps in the search space, as in Othello, or in the case of MCTS-MR a relatively high branching factor, as in Breakthrough. In future work, these problems could be addressed with the help of domain knowledge. On the one hand, domain knowledge could be incorporated into the hybrid algorithms in the form of *evaluation functions*. This could make minimax potentially much more useful in search spaces with few terminal nodes before the latest game phase, such as that of Othello or Go. The use of heuristic evaluation functions could make it possible for MCTS-minimax hybrids to not only detect the type of traps studied in this article—traps that lead to a lost game—but also *soft traps* that only lead to a disadvantageous position [19]. The main challenge for this approach is properly combining results of heuristic evaluation functions with the results of rollout returns, their averages and confidence intervals, as produced by MCTS. One could also conceive of hybrids using the rollout returns themselves as evaluation function. On the other hand, domain knowledge could be incorporated in the form of a *move ordering function*. This could be effective in games such as Breakthrough, where traps are relatively frequent, but the branching factor seems to be too high for MCTS-MR. Here, the embedded minimax searches could only take the highest-ranked moves into account, reducing their overhead. Such a *k-best* approach has successfully been applied to MCTS-MR in other games [18], [20]. In domains with low density of terminal nodes however, move ordering could be potentially less effective for MCTS-minimax hybrids without heuristic evaluation functions because most leaf nodes of the embedded searches are non-terminal positions and therefore equal in value.

Note that in all experiments except those of Subsections V-D1 and V-C1, we used fast, uniformly random rollout policies. On the one hand, the overhead of our techniques would be proportionally lower for any slower, informed rollout policies such as typically used in state-of-the-art programs. On the other hand, improvement on already strong policies might prove to be more difficult. Examining the influence of such MCTS implementation properties is a possible second direction of future research.

Third, while we have focused primarily on the game tree properties of trap density and difficulty as well as the average branching factor in this article, the impact of other properties such as the game length or the distribution of terminal values also deserve further study. Artificial game trees could be used to study these properties in isolation—the large number of differences between "real" games potentially confounds many effects, such as Breakthrough featuring more traps throughout the search space than Othello, but also having a larger branching factor. Eventually, it might be possible to learn from the success of MCTS-minimax hybrids in Catch the Lion, and transfer some ideas to larger games of similar type such as Shogi and chess.

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] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *5th International Conference on Computers and Games (CG 2006). Revised Papers*, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer, 2007, pp. 72–83.

[2] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *17th European Conference on Machine Learning, ECML 2006*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer, 2006, pp. 282–293.

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

[4] R. J. Lorentz, "Amazons Discover Monte-Carlo," in *6th International Conference on Computers and Games, CG 2008*, ser. Lecture Notes in Computer Science, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., vol. 5131. Springer, 2008, pp. 13–24.

[5] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte Carlo Tree Search in Lines of Action," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 239–250, 2010.

[6] R. Ramanujan, A. Sabharwal, and B. Selman, "On Adversarial Search Spaces and Sampling-Based Planning," in *20th International Conference on Automated Planning and Scheduling, ICAPS 2010*, R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz, Eds. AAAI, 2010, pp. 242–245.

[7] H. Baier and M. H. M. Winands, "Monte-Carlo Tree Search and Minimax Hybrids," in *2013 IEEE Conference on Computational Intelligence and Games, CIG 2013*, 2013, pp. 129–136.

[8] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

[9] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-Time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[10] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[11] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte-Carlo Tree Search Solver," in *6th International Conference on Computers and Games, CG 2008*, ser. Lecture Notes in Computer Science, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., vol. 5131. Springer, 2008, pp. 25–36.

[12] R. Ramanujan and B. Selman, "Trade-Offs in Sampling-Based Adversarial Planning," in *21st International Conference on Automated Planning and Scheduling, ICAPS 2011*, F. Bacchus, C. Domshlak, S. Edelkamp, and M. Helmert, Eds. AAAI, 2011.

[13] R. Ramanujan, A. Sabharwal, and B. Selman, "On the Behavior of UCT in Synthetic Search Spaces," in *ICAPS 2011 Workshop on Monte-Carlo Tree Search: Theory and Applications*, 2011.

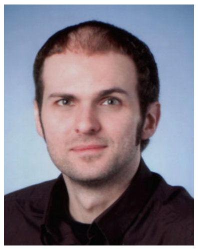
[14] H. Finnsson and Y. Björnsson, "Game-Tree Properties and MCTS Performance," in *IJCAI'11 Workshop on General Intelligence in Game Playing Agents (GIGA'11)*, 2011, pp. 23–30.

[15] J. E. Clune, "Heuristic Evaluation Functions for General Game Playing," Ph.D. dissertation, Department of Computer Science, University of California, Los Angeles, USA, 2008.

[16] F. Teytaud and O. Teytaud, "On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms," in *2010 IEEE Conference on Computational Intelligence and Games, CIG 2010*, G. N. Yannakakis and J. Togelius, Eds. IEEE, 2010, pp. 359–364.

[17] R. J. Lorentz, "Experiments with Monte-Carlo Tree Search in the Game of Havannah," *ICGA Journal*, vol. 34, no. 3, pp. 140–149, 2011.

[18] M. H. M. Winands and Y. Björnsson, "Alpha-Beta-based Play-outs in Monte-Carlo Tree Search," in *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011*, S.-B. Cho, S. M. Lucas, and P. Hingston, Eds. IEEE, 2011, pp. 110–117.
[19] R. Ramanujan, A. Sabharwal, and B. Selman, "Understanding Sampling Style Adversarial Search Methods," in *26th Conference on Uncertainty in Artificial Intelligence, UAI 2010*, P. Grünwald and P. Spirtes, Eds., 2010, pp. 474–483.
[20] J. A. M. Nijssen and M. H. M. Winands, "Playout Search for Monte-Carlo Tree Search in Multi-player Games," in *13th International Conference on Advances in Computer Games, ACG 2011*, ser. Lecture Notes in Computer Science, H. J. van den Herik and A. Plaat, Eds., vol. 7168. Springer, 2012, pp. 72–83.
[21] T. Cazenave, "Nested Monte-Carlo Search," in *21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, C. Boutilier, Ed., 2009, pp. 456–461.
[22] H. Baier and M. H. M. Winands, "Nested Monte-Carlo Tree Search for Online Planning in Large MDPs," in *20th European Conference on Artificial Intelligence, ECAI 2012*, ser. Frontiers in Artificial Intelligence and Applications, L. De Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, Eds., vol. 242. IOS Press, 2012, pp. 109–114.
[23] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with Patterns in Monte-Carlo Go," HAL - CCSd - CNRS, France, Tech. Rep., 2006.
[24] Y. Sato, D. Takahashi, and R. Grimbergen, "A Shogi Program Based on Monte-Carlo Tree Search," *ICGA Journal*, vol. 33, no. 2, pp. 80–92, 2010.
[25] B. Bouzy and G. M. J.-B. Chaslot, "Monte-Carlo Go Reinforcement Learning Experiments," in *2006 IEEE Symposium on Computational Intelligence and Games, CIG 2006*, G. Kendall and S. Louis, Eds., 2006, pp. 187–194.
[26] S. Gelly and D. Silver, "Combining Online and Offline Knowledge in UCT," in *24th International Conference on Machine Learning, ICML 2007*, ser. ACM International Conference Proceeding Series, Z. Ghahramani, Ed., vol. 227, 2007, pp. 273–280.



Hendrik Baier holds a B.Sc. in Computer Science (Darmstadt Technical University, Darmstadt, Germany, 2006), an M.Sc. in Cognitive Science (Osnabrück University, Osnabrück, Germany, 2010), and is currently completing a Ph.D. in Artificial Intelligence (Maastricht University, Maastricht, The Netherlands).

Mark Winands received a Ph.D. degree in Artificial Intelligence from the Department of Computer Science, Maastricht University, Maastricht, The Netherlands, in 2004.

Currently, he is an Assistant Professor at the Department of Knowledge Engineering, Maastricht University. His research interests include heuristic search, machine learning and games.

Dr. Winands serves as a section editor of the ICGA Journal and as an associate editor of IEEE Transactions on Computational Intelligence and AI in Games.