# MCTS-Minimax Hybrids with State Evaluations

**Hendrik Baier** hendrik.baier@york.ac.uk
*Digital Creativity Labs,*
*University of York, York, UK*
**Mark H. M. Winands** m.winands@maastrichtuniversity.nl
*Games and AI Group, Department of Data Science and Knowledge Engineering,*
*Maastricht University, Maastricht, The Netherlands*

## Abstract

*Monte-Carlo Tree Search* (MCTS) has been found to show weaker play than minimax-based search in some tactical game domains. This is partly due to its highly selective search and averaging value backups, which make it susceptible to traps. 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 article 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. Furthermore, all three hybrids are enhanced with the help of move ordering and *k*-best pruning for minimax. Results show that the use of enhanced minimax for computing node priors results in the strongest MCTS-minimax hybrid investigated in the three test domains of Othello, Breakthrough, and Catch the Lion. This hybrid, called MCTS-IP-M-k, also outperforms enhanced minimax as a standalone player in Breakthrough, demonstrating that at least in this domain, MCTS and minimax can be combined to an algorithm stronger than its parts. Using enhanced minimax for computing node priors is therefore a promising new technique for integrating domain knowledge into an MCTS framework.

## 1. Introduction

Monte-Carlo Tree Search (MCTS) (Coulom, 2007; Kocsis & Szepesvári, 2006) is a best-first tree search algorithm based on Monte-Carlo simulations for state evaluation. It selectively samples actions instead of expanding all legal actions from a given state, which allows it to handle large search spaces with high branching factors. Monte-Carlo simulations make it independent of a static heuristic evaluation function when comparing non-terminal states. If exploration and exploitation are traded off appropriately, MCTS asymptotically converges to the optimal policy (Kocsis & Szepesvári, 2006), while providing approximations at any time.

MCTS has shown considerable success in a variety of domains—see Silver et al. (2016) for the prominent recent example of Google DeepMind's AlphaGo, and Browne et al. (2012) for an earlier literature survey. However, there are still a number of adversarial domains such as the games of Chess and (International) Checkers in which the traditional approach to adversarial planning, minimax search with $\alpha\beta$ pruning (Knuth & Moore, 1975), remains superior. 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 terminal states and *shallow traps* exist in the search space (Ramanujan, Sabharwal, & Selman, 2010a). These require precise play to avoid immediate loss, and the selective sampling and averaging value backups of MCTS can easily miss or underestimate an important move. 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. Here, MCTS can fully play out its strategic and positional understanding resulting from Monte-Carlo simulations of entire games.

Note that the recent publication of AlphaZero (Silver et al., 2017b) points towards MCTS possibly becoming the dominant approach even for Chess-like games. Deep reinforcement learning seems to be able to produce state evaluators that avoid traps surprisingly well on their own. Regardless of the state evaluator however, MCTS is still more susceptible to traps than alpha-beta, and deep neural networks may give a boost to alpha-beta as well. While AlphaZero opens new questions, this article could give an indication as to how the strengths of MCTS and alpha-beta can be combined using any evaluation function, hand-coded or learned.

In previous work (Baier & Winands, 2015), *MCTS-minimax hybrids* have been introduced, embedding shallow minimax searches into the MCTS framework. This was a first step towards combining the strategic strength of MCTS with the tactical strength of minimax. The results of the hybrid algorithms MCTS-MR, MCTS-MS, and MCTS-MB have been promising even without making use of domain knowledge such as heuristic evaluation functions. However, their inability to evaluate non-terminal states makes them ineffective in games with very few or no terminal states throughout most of the search space, such as the game of Othello. Furthermore, some form of state evaluation is often available in practice—as AlphaGo (Silver et al., 2016) demonstrated, deep learning makes this possible even for domains where hand-crafting evaluation functions has traditionally been considered very difficult. This article therefore continues this line of research by addressing the case where domain knowledge is available.

The algorithms discussed in this article make use of *state evaluations*. These state evaluations can either be the result of simple evaluation function calls, or the result of minimax searches using the same evaluation function at the leaves. Three different approaches for integrating state evaluations into MCTS are considered. The first approach uses state evaluations to choose rollout moves (MCTS-*IR* for *informed rollouts*). The second approach uses state evaluations to terminate rollouts early (MCTS-*IC* for *informed cutoffs*). The third approach uses state evaluations to bias the selection of moves in the MCTS tree (MCTS-*IP* for *informed priors*). Using minimax with $\alpha\beta$ to compute state evaluations means accepting longer computation times in favor of typically more accurate evaluations as compared to simple evaluation function calls. Only in the case of MCTS-IR, minimax has been applied before (Ramanujan, Sabharwal, & Selman, 2010b); the use of minimax for the other two approaches is newly proposed in the form described here. 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.

After the branching factor of a domain is identified as a limiting factor of the hybrids' performance, further experiments are conducted using domain knowledge not only for state evaluation, but also for *move ordering*. Move ordering reduces the average size of $\alpha\beta$ trees, and furthermore allows to restrict the effective branching factor of $\alpha\beta$ to only the $k$ most promising moves in any given state (*k-best pruning*). Again, this has only been done for MCTS with minimax rollouts before (Winands, Björnsson, & Saito, 2010). The enhanced MCTS-minimax hybrids with move ordering and $k$-best pruning are tested and compared to the unenhanced hybrids as well as the equivalent algorithms using static evaluations in all three domains. They are also tested against each other to determine the relative strongest hybrid, compared across domains, studied at different time settings and different branching factors, combined with each other, and finally compared to an $\alpha\beta$ baseline.

This article is an extension of Baier and Winands (2014). It explains how to overcome the problems identified there with the help of move ordering and $k$-best pruning. The resulting algorithms MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k are introduced, tested and evaluated in the newly added Subsection 5.2, leading to the novel contribution of MCTS-IP-M-k as a promising technique for integrating domain knowledge into an MCTS framework.

This article is structured as follows. Section 3 provides a brief overview of related work 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 embedded minimax searches for each of these. Two of these MCTS-minimax hybrids are newly proposed here. 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

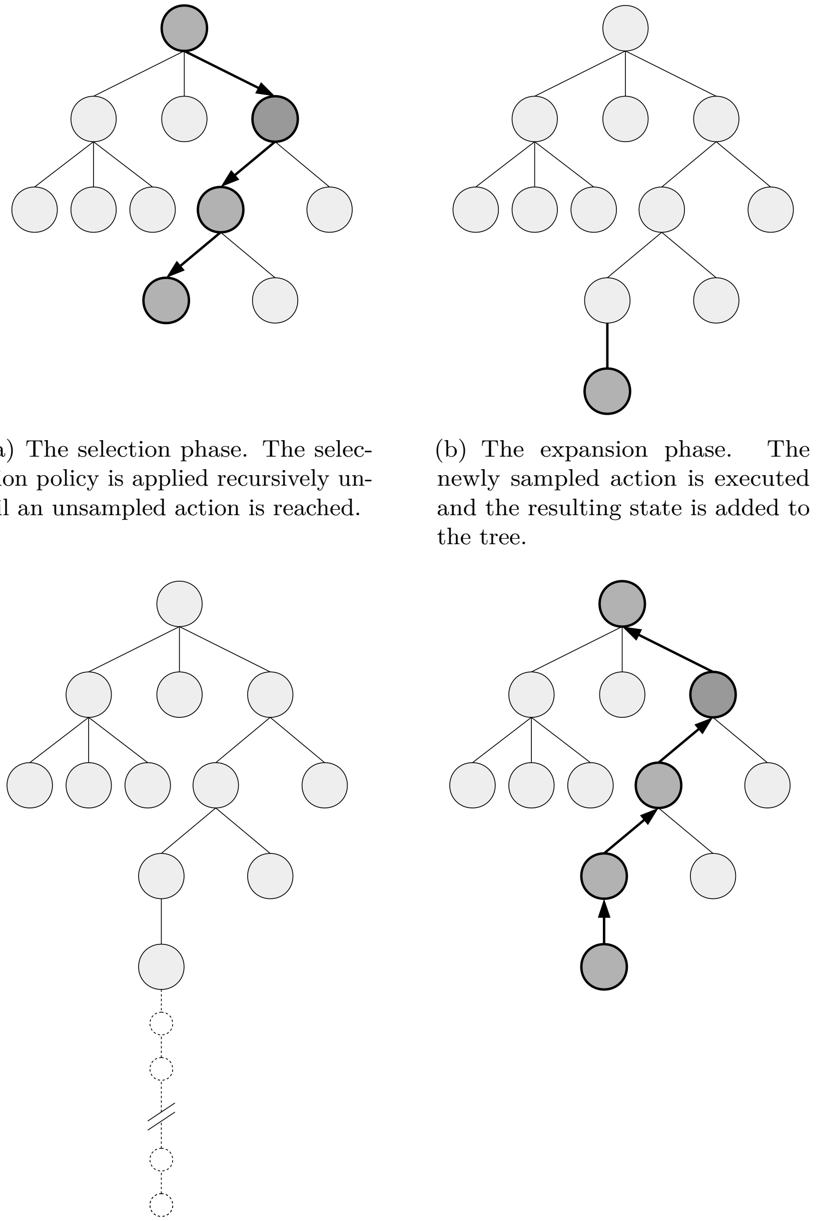
Monte-Carlo Tree Search (MCTS) is the underlying framework of the algorithms in this article. MCTS works by repeating the following four-phase loop until computation time runs out (Chaslot et al., 2008). The root of the tree represents the current state of the game. Each iteration of the loop represents one simulated game. The phases are visualized in Figure 1.

Phase one: *selection.* The tree is traversed from the root to one of its leaves, 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 UCB1-TUNED policy is used as selection policy (Auer, Cesa-Bianchi, & Fischer, 2002).

Phase two: *expansion.* When a leaf has been reached, 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 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.



(a) The selection phase. The selection policy is applied recursively until an unsampled action is reached.

(b) The expansion phase. The newly sampled action is executed and the resulting state is added to the tree.

(c) The rollout phase. One simulated game is played by the rollout policy.

(d) The backpropagation phase. The result of the rollout is used to update value estimates in the tree.

Figure 1: MCTS.

Many variants and extensions of this framework have been proposed in the literature (Browne et al., 2012). In this article, we are using MCTS with the *MCTS-Solver* extension (Winands, Björnsson, & Saito, 2008) 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

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 Ramanujan and Selman (2011), where the algorithm $UCTMAX_H$ replaces MCTS rollouts with heuristic evaluations and classic averaging MCTS backups with minimaxing backups. In *implicit minimax backups* (Lanctot et al., 2014), 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 this work.

The idea of improving Monte-Carlo rollouts with the help of heuristic domain knowledge has first been applied to games by Bouzy (2005). 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 Lorentz (2011) for the game of Havannah, or a 2-ply search for Lines of Action (Winands & Björnsson, 2011), Chess (Ramanujan et al., 2010b), and multi-player games (Nijssen & Winands, 2012). 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 (Lorentz, 2008), Lines of Action (Winands et al., 2010), and Breakthrough (Lorentz & Horey, 2014). It was first described in a naive Monte Carlo context (without tree search) by Sheppard (2002). 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 Lorentz (2008) and Winands et al. (2010) as we backpropagate the actual heuristic values instead of rounding them to losses or wins. They are also different from Winands et al. (2010) 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 Gelly and Silver (2007) and Chaslot et al. (2008) for the game of Go. Our implementation is similar to Gelly and Silver (2007) 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.

Recent work on AlphaGo Zero (Silver et al., 2017a) and AlphaZero (Silver et al., 2017b) has demonstrated the impressive success of using deep reinforcement learning to train a neural network which encodes domain knowledge. This network was then used within MCTS both for biasing the selection policy as well as for replacing the rollout policy with a state evaluation. It remains an interesting line of future work to see if powerful function approximators such as deep neural networks can also be even more effectively used in hybrid search algorithms such as those proposed here.

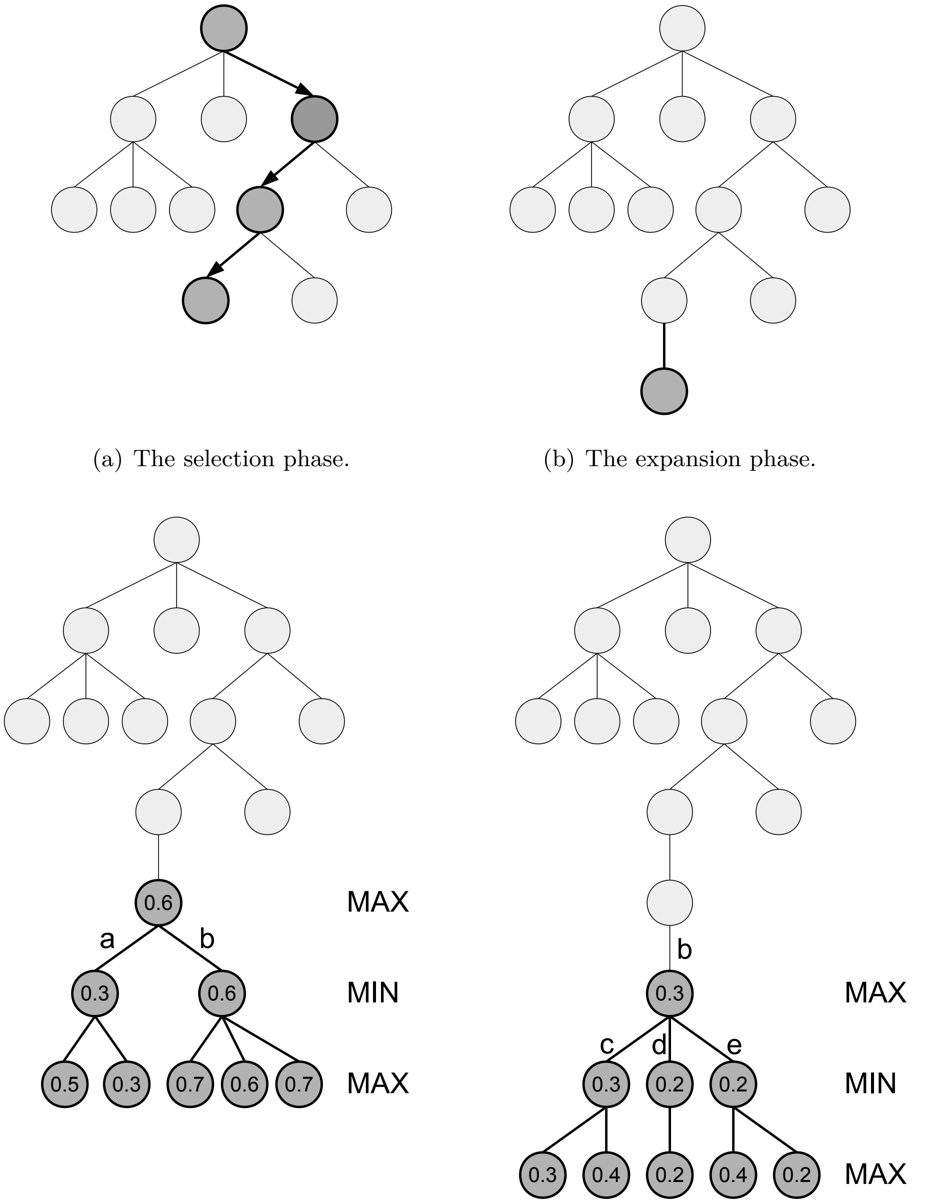
## 4. Hybrid Algorithms

As described in Section 2, MCTS-Solver is used as the baseline. This section describes the three different approaches for employing heuristic knowledge within MCTS that we explore in this article. 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 (Gelly et al., 2006), and preserve convergence if they explore sufficiently. 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 $\epsilon$, 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 *$\epsilon$-greedy* (Sturtevant, 2008). 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 $\epsilon$-greedy policy can be extended in a natural way to a depth-$d$ minimax search for every rollout move (Winands & Björnsson, 2011; Nijssen & Winands, 2012). We use a random move ordering in minimax as well in order to preserve non-determinism. In contrast to Winands and Björnsson (2011) and Nijssen and Winands (2012) where several enhancements such as move ordering, $k$-best pruning, and killer moves were added to $\alpha\beta$, we first test unenhanced $\alpha\beta$ search in Subsection 5.2. 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 Winands and Björnsson (2011) and Nijssen and Winands (2012), respectively. Move ordering and $k$-best pruning are then added in Subsection 5.3. 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). An example is visualized in Figure 2.



(a) The selection phase.

(b) The expansion phase.

(c) A $d = 2$ minimax search is started to find the first rollout move. The maximizing player chooses move $b$ with a heuristic value of 0.6.

(d) Another $d = 2$ minimax search is conducted for the second rollout move. In this case, the maximizing player chooses move $c$ with a heuristic value of 0.3.

Figure 2: The MCTS-IR-M hybrid. $\epsilon = 0$ and $d = 2$.

### 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 (Lorentz, 2008; Winands et al., 2010). 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 rollout 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 $[0, 1]$. 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 Winands and Björnsson (2011), 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 rollout returns, but by computing them directly instead of playing out the rollout. We call this approach *MCTS-IC-M* (MCTS with informed cutoffs using minimax). An example is shown in Figure 3. Subsections 5.2.3 and 5.3.4 present results on the performance of this strategy using $\alpha\beta$ in unenhanced form and with $\alpha\beta$ using move ordering and $k$-best pruning, respectively.

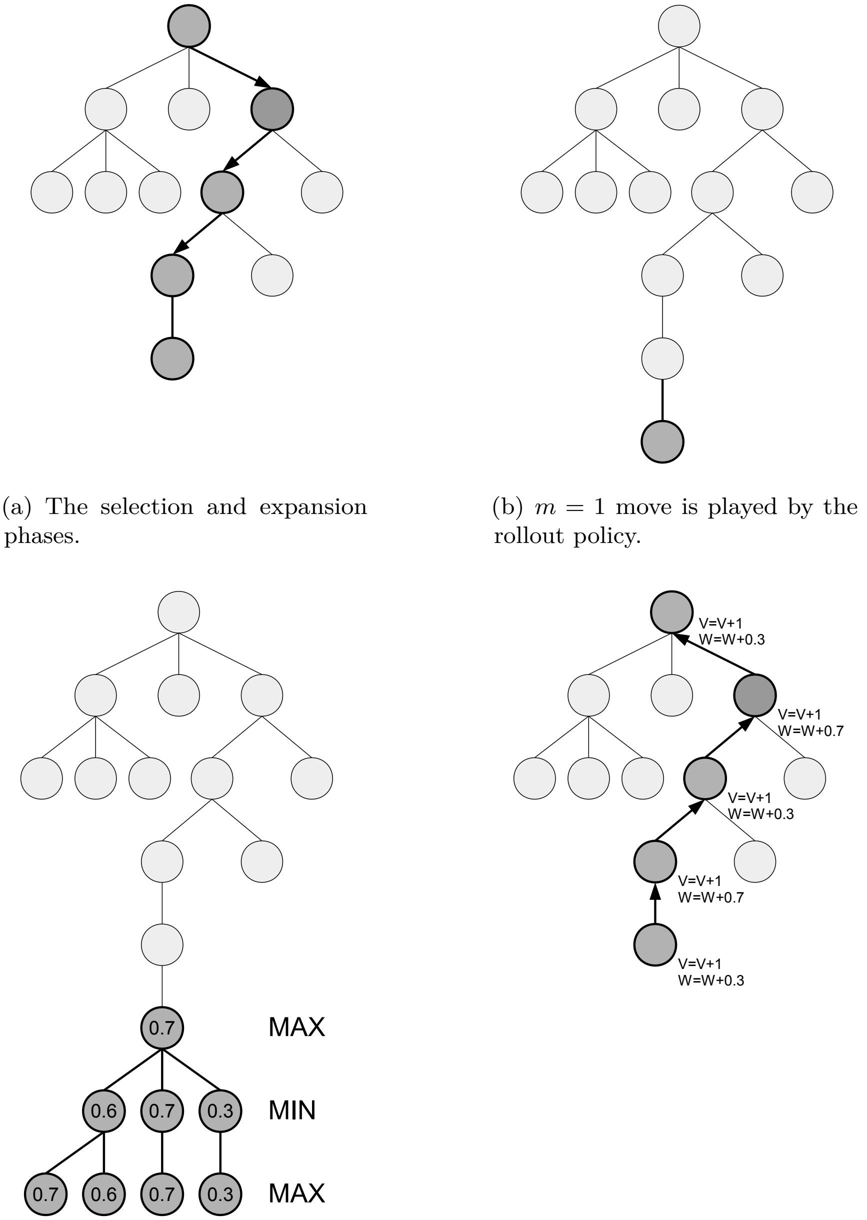
### 4.3 MCTS with Informed Priors (MCTS-IP)

Node priors (Gelly & Silver, 2007) 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 $h$ 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 $\gamma$. The following formulas show how to update the win ($w$) and visit ($v$) counters of the node at hand.

$$v \longleftarrow v + \gamma \tag{1a}$$

$$w \longleftarrow w + \gamma h \tag{1b}$$

We assume $h \in [0, 1]$. If the evaluation value $h$ is 0.6 and the weight $\gamma$ is 100, for example, 60 wins and 100 visits are added to the node at hand. This is equivalent to 60 virtual wins and $100 - 60 = 40$ virtual losses. 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



(a) The selection and expansion phases.

(b) $m = 1$ move is played by the rollout policy.

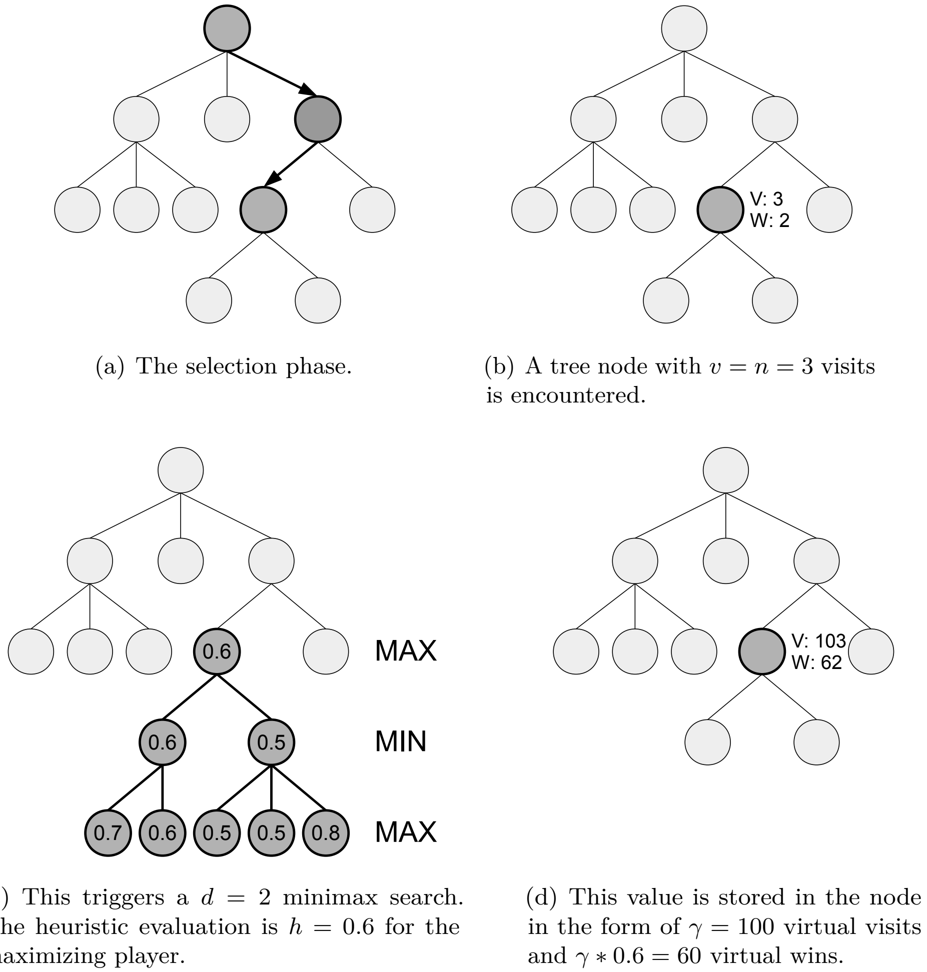
(c) The resulting position is evaluated with a $d = 2$ minimax search. The heuristic evaluation is 0.7 for the maximizing player.

(d) This value is backpropagated as rollout return. Each traversed node increments its visit count by 1, and its win count by 0.7 or $1-0.7 = 0.3$ depending on the player to move.

Figure 3: The MCTS-IC-M hybrid. $m = 1$ and $d = 2$.

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 article.

We propose to extend MCTS-IP with a depth-$d$ minimax search in order to compute the prior value to be stored. This approach aims at guiding the selection policy through more accurate prior information in the nodes of the MCTS tree. We call this approach *MCTS-IP-M* (MCTS with informed priors using minimax). See Figure 4 for an illustration. Just like the other hybrids, we first test MCTS-IP-M with $\alpha\beta$ in unenhanced form (Subsection 5.2.4), and then introduce move ordering and $k$-best pruning (Subsection 5.3.5).



(a) The selection phase.

(b) A tree node with $v = n = 3$ visits is encountered.

(c) This triggers a $d = 2$ minimax search. The heuristic evaluation is $h = 0.6$ for the maximizing player.

(d) This value is stored in the node in the form of $\gamma = 100$ virtual visits and $\gamma * 0.6 = 60$ virtual wins.

Figure 4: The MCTS-IP-M hybrid. $n = 3$, $d = 2$, and $\gamma = 100$.

## 5. Experimental Results

We tested the algorithms in three different domains: *Othello*, *Catch the Lion*, and *6×6 Breakthrough*. These are all deterministic perfect-information turn-taking zero-sum games. Othello, closely related to *Reversi*, is a territory game which has been subject to research both in the minimax framework (Rosenbloom, 1982) as well as for MCTS in General Game Playing (Finnsson & Björnsson, 2008). Its three- and four-player variant *Rolit* has also been used for investigating multi-player search algorithms (Schadd & Winands, 2011). Catch the Lion—or *doubutsu shogi*, "animal Chess" in Japanese—is a capture game developed by professional Shogi (Japanese Chess) players Madoka Kitao and Maiko Fujita in 2008 in order to attract children to Shogi. It attempts to reflect all essential characteristics of Shogi in the simplest possible form (see Sato, Takahashi, and Grimbergen (2010) for an MCTS approach to the full game of Shogi). It is used here because it represents a simple instance of Chess-like games, which tend to be particularly difficult for MCTS (Ramanujan et al., 2010a). Breakthrough is a race game invented by Dan Troyka in 2000 for a game design competition. The application of MCTS to $(8 \times 8)$ Breakthrough has been investigated by Lorentz and Horey (2014). Breakthrough is also a popular domain in the General Game Playing community (Finnsson & Björnsson, 2008; Tak, Winands, & Björnsson, 2012; Gudmundsson & Björnsson, 2013). The game was originally described as being played on a $7 \times 7$ board, but other sizes such as $8 \times 8$ are popular as well, and the $6 \times 6$ board used here preserves an interesting search space.

This section is organized as follows. Subsection 5.1 outlines the evaluation functions used for each game. Next, Subsection 5.2 presents experimental results for MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E, as well as MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M using unenhanced $\alpha\beta$. After identifying the branching factor of a domain as an important limiting factor for algorithm performance, Subsection 5.3 then provides improved results for MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k using $\alpha\beta$ with move ordering and $k$-best pruning, and analyzes these hybrids more deeply. Both Subsection 5.2 and 5.3 begin with a brief summary of the main takeaways in order to improve readability.

### 5.1 Evaluation Functions

This subsection outlines the heuristic board evaluation functions used for each of the three test domains. 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 evaluation function we use for Othello is adapted from the Rolit evaluation function described in Nijssen (2013). It first determines the number of *stable* discs for the player, i.e. discs that cannot change color anymore for the rest of the game. For each stable disc of her color, the player receives 10 points. Afterwards, the number of legal moves for the player is added to the score in order to reward mobility.

**Catch the Lion.** The evaluation function 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.

**Breakthrough.** The evaluation score we use for 6×6 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. This evaluation function is a simplified version of the one used in Lorentz and Horey (2014).

### 5.2 Results with Unenhanced $\alpha\beta$

In all experimental conditions, we compared the hybrids (ending -M) as well as their counterparts using heuristics without minimax (ending -E) against regular MCTS-Solver as the baseline. Rollouts were uniformly random except for MCTS-IR. 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 set to 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. In order to make a comparison of the relative computational costs of all algorithms possible, we are also listing their speed in terms of rollouts per second (rps), measured on an Intel Core i7-3667U CPU at 2.0 GHz using the average over 10 1-second searches from the initial board position in each game. Our MCTS-Solver baseline searches with 9000 rps in Othello, 73600 rps in Breakthrough, and 66900 rps in Catch the Lion.

Subsection 5.2.1 briefly summarizes the main findings. Subsections 5.2.2 to 5.2.4 present in more detail the results for MCTS-IR, MCTS-IC, and MCTS-IP, respectively.

#### 5.2.1 Main Findings

- MCTS-IR-E works well in all three domains. However, MCTS-IR-M does not improve on it in any domain. The reason is the high computational cost of calling minimax many times in each MCTS simulation.

- MCTS-IC-E works well in all domains but Breakthrough (the evaluation function we used in Breakthrough may not be accurate enough for MCTS to fully rely on it instead of rollouts). However, MCTS-IC-M does not improve on MCTS-IC-E in any domain, the reason again being the high computational cost of calling minimax even just once in every MCTS simulation.

- MCTS-IP-E works well in all domains. MCTS-IP-M improves on it in all domains but Breakthrough. The reason is probably that the parameters of MCTS-IP make it easier to control the computational cost, but the branching factor of Breakthrough (15.5 compared to 10.5 in Catch the Lion and 8 in Othello) is still causing problems.

- In conclusion, MCTS-IP-M seems to be a promising hybrid, but the sensitivity to the branching factor of the given domain is a problem for the hybrids in general. Additional steps need to be taken to reduce the effective branching factor.

### 5.2.2 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 5(a) to 5(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 ($p<0.001$).

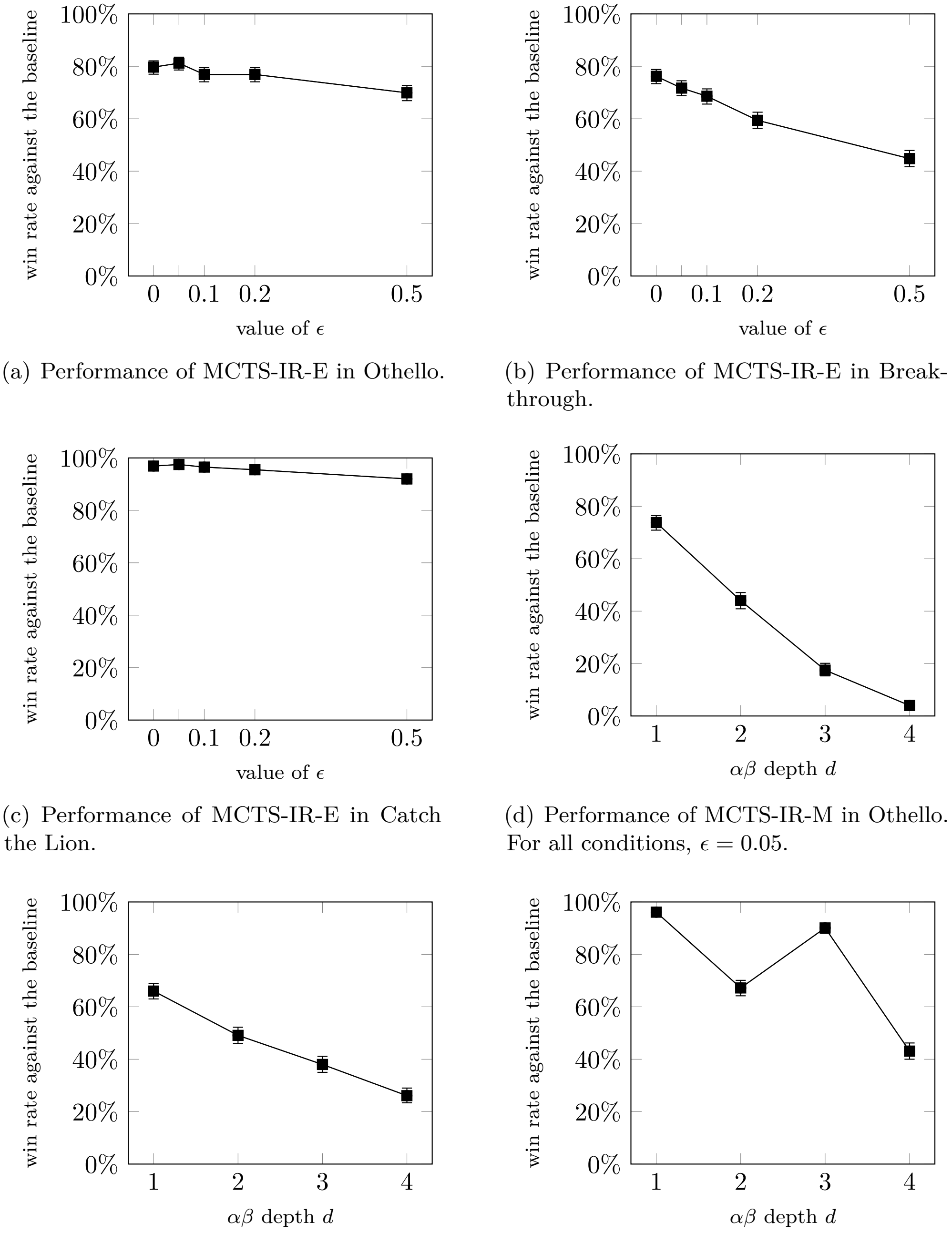
MCTS-IR-M was tested for $d \in \{1, \ldots, 4\}$ with the optimal value of $\epsilon$ found for each domain in the MCTS-IR-E experiments. Each condition played 1000 games per domain against the baseline player. The results are presented in Figures 5(d) to 5(f). The most promising setting in all domains 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.

The best-performing settings of MCTS-IR-E searched with 370 rps in Othello, 3800 rps in Breakthrough, and 34500 rps in Catch the Lion. MCTS-IR-M achieved 320 rps in Othello, 2300 rps in Breakthrough, and 30100 rps in Catch the Lion.

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% (47.3%-53.5%) 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 (compare Gelly and Silver (2007), Silver and Tesauro (2009)).



(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, $\epsilon = 0.05$.

Figure 5: Performance of MCTS-IR in Othello, Breakthrough and Catch the Lion.

### 5.2.3 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 6(a) to 6(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 subsection, we tuned against regular MCTS-Solver only, and both MCTS-Solver and MCTS-IC used uniformly random rollouts.

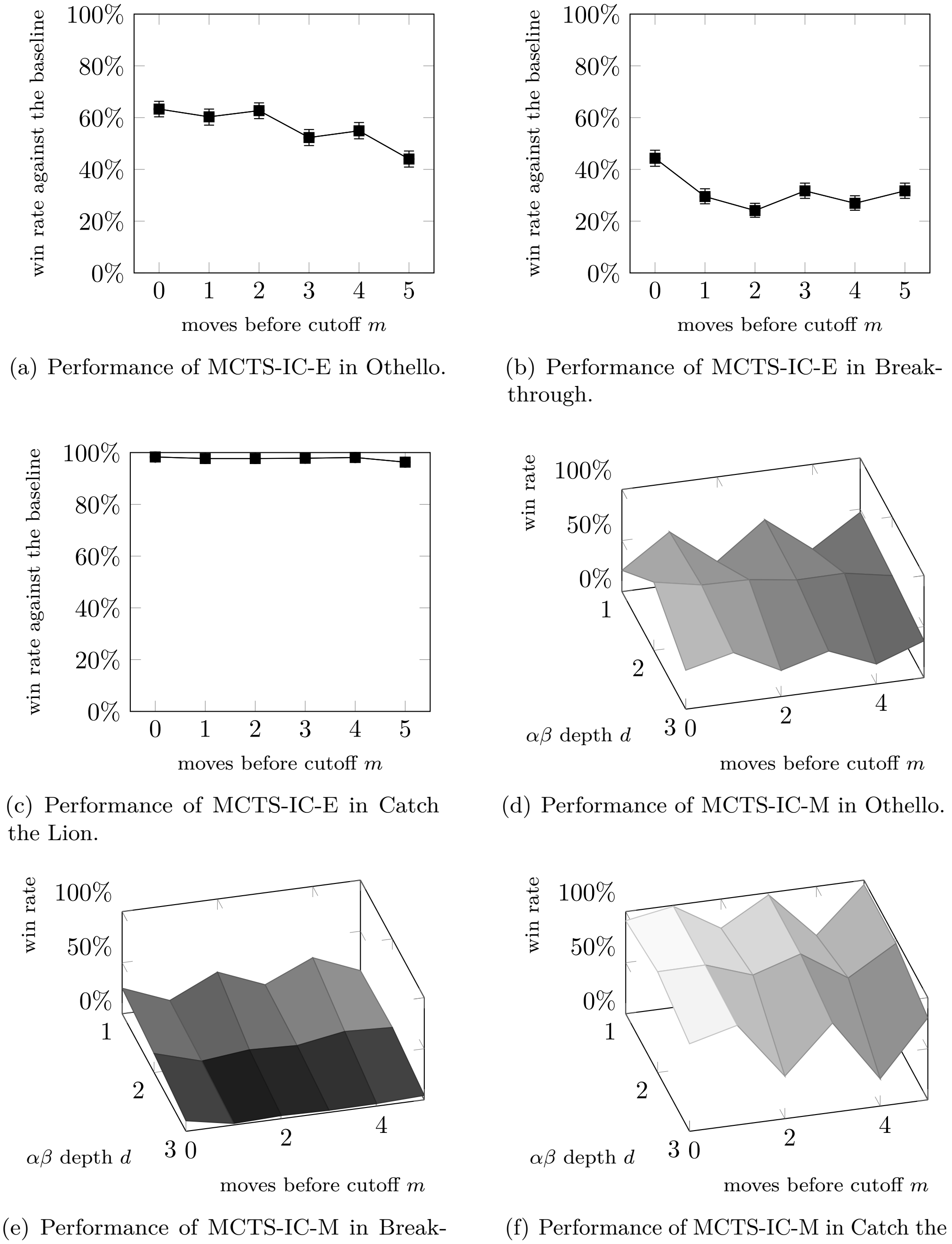
MCTS-IC-M was tested for all combinations of $m \in \{0, \ldots, 5\}$ and $d \in \{1, 2, 3\}$, with 1000 games each per domain. The results are shown in Figures 6(d) to 6(f). The best performance was achieved with $m = 0$ and $d = 2$ in Othello, $m = 4$ and $d = 1$ in Breakthrough, and $m = 1$ and $d = 2$ 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 against MCTS-IC-M 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 MCTS-IC-E (p<0.001). This confirms that MCTS-IC-M is weaker than MCTS-IC-E due to its time overhead.

The best-performing settings of MCTS-IC-E searched with 109200 rps in Othello, 153500 rps in Breakthrough, and 133700 rps in Catch the Lion. Note the speedup due to not having to perform rollouts, in particular for games where computing legal moves is more complex, such as Othello. MCTS-IC-M achieved 11600 rps in Othello, 47600 rps in Breakthrough, and 41200 rps in Catch the Lion.

A seemingly paradoxical observation was made with MCTS-IC as well. In 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 14.8% of 1000 test games against MCTS-IC-E with $m = 0$, at 10000 rollouts per move. With $d = 2$, it won 38.2%. Even with $d = 3$, it won only 35.9% (all at 10000 rollouts per move). Once more these results demonstrate that a stronger policy can lead to a weaker search when embedded in MCTS.



(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.

Figure 6: Performance of MCTS-IC in Othello, Breakthrough and Catch the Lion.

### 5.2.4 Experiments with MCTS-IP

MCTS-IP-E was tested for all combinations of $n \in \{0, 1, 2\}$ and $\gamma \in \{50, 100, 250, 500, 1000, 2500, 5000\}$. Each condition played 1000 games per domain against the baseline player. The results are shown in Figures 7(a) to 7(c). The best-performing conditions were $n = 1$ and $\gamma = 1000$ in Othello, $n = 1$ and $\gamma = 2500$ in Breakthrough, and $n = 0$ and $\gamma = 100$ 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 $n \in \{0, 1, 2, 5, 10, 25\}$, $\gamma \in \{50, 100, 250, 500, 1000, 2500, 5000\}$, and $d \in \{1, \ldots, 5\}$ with 1000 games per condition in each domain. Figures 7(d) to 7(f) present the results, using the optimal setting of $d$ for all domains. The most promising parameter values found in Othello were $n = 2$, $\gamma = 5000$, and $d = 3$. In Breakthrough they were $n = 1$, $\gamma = 1000$, and $d = 1$, and in Catch the Lion they were $n = 1$, $\gamma = 2500$, 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 most likely due to the larger branching factor of Breakthrough slowing MCTS-IP-M down—it has a branching factor of 15.5 on average, while Catch the Lion has 10.5 and Othello has 8. At 1000 rollouts per move, MCTS-IP-M with $n = 1$, $\gamma = 1000$, and $d = 1$ won 91.1% of 1000 games against the best MCTS-IP-E setting in Breakthrough as well.

An interesting observation is the high weight 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 due to the tactical nature of the game. Exchanges of pieces can often lead to quick and drastic changes of the evaluation values, or even to the end of the game by capturing a Lion (see Baier and Winands (2015) for a comparison of the domains' tacticality in terms of trap density and trap difficulty). 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 subsection.

The best-performing settings of MCTS-IP-E searched with 8800 rps in Othello, 68600 rps in Breakthrough, and 58000 rps in Catch the Lion. MCTS-IP-M achieved 2500 rps in

Othello, 43700 rps in Breakthrough, and 870 rps in Catch the Lion. Note how in the tactical Catch the Lion, more accurate priors are worth giving up a large percentage of rollouts.

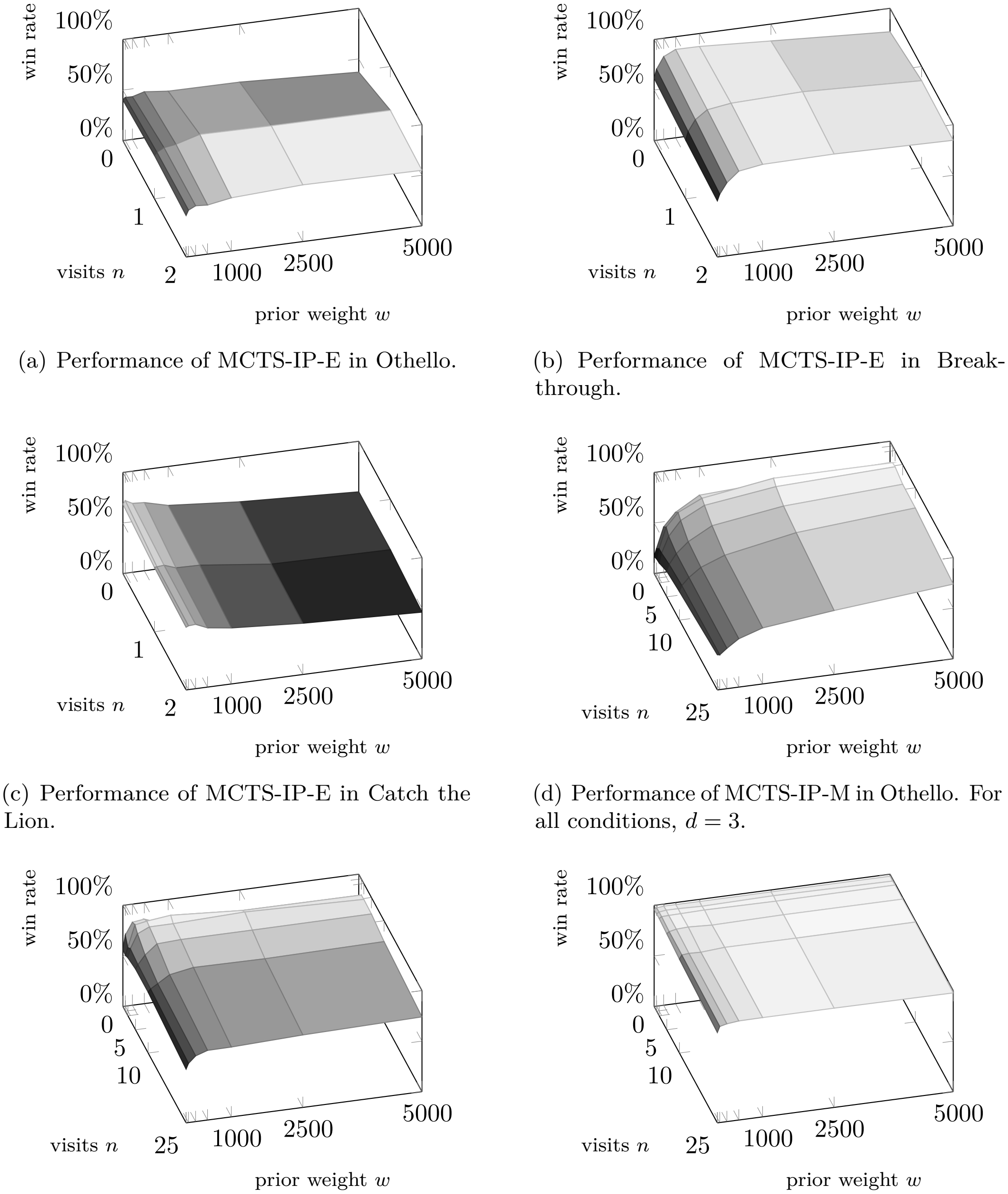
The effect of stronger policies resulting in weaker performance when integrated into MCTS can be found in MCTS-IP just as in MCTS-IR and MCTS-IC. In Breakthrough for example, MCTS-IP-M with $n = 1$, $\gamma = 1000$, and $d = 2$ won only 83.4% of 1000 games against the strongest MCTS-IP-E setting, compared to 91.1% with $n = 1$, $\gamma = 1000$, 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 our Breakthrough experiments.

### 5.3 Results with Move Ordering and $k$-best Pruning

In the previous subsection, $\alpha\beta$ search was used in its basic, unenhanced form. This was sufficient to improve MCTS-IP in Othello and Catch the Lion, but too computationally expensive for MCTS-IR and MCTS-IC in these two domains, as well as for all hybrids in Breakthrough. The performance difference between the hybrids can be explained by the fact that MCTS-IP allows to control the frequency of minimax calls with the parameter $n$, while MCTS-IC needs to call minimax once in every rollout, and MCTS-IR even for every single rollout move. This makes it easier for MCTS-IP to trade off the computational cost of embedded minimax searches against their advantages over static evaluation function calls. In previous work on a combination of MCTS-IR and MCTS-IC (Winands & Björnsson, 2011), the computational cost seemed to be have less impact due to the longer time settings of up to 30 seconds per move. A reduction of the number of rollouts is less problematic at long time settings because rollouts can bring diminishing returns (see e.g. Robilliard, Fonlupt, and Teytaud (2014)). The performance difference between the domains, however, can be explained by the larger branching factor of 6×6 Breakthrough compared to Othello and Catch the Lion, which affects full-width minimax more strongly than for example the sampling-based MCTS. The main problem of MCTS-minimax hybrids seems to be their sensitivity to the branching factor of the domain.

In this subsection, we therefore conduct further experiments applying limited domain knowledge not only for state evaluation, but also for move ordering. The application of move ordering is known to strongly improve the performance of $\alpha\beta$ through a reduction of the average size of the search tree (Knuth & Moore, 1975). Additionally, with a good move ordering heuristic one can restrict $\alpha\beta$ to only searching the $k$ moves in each state that seem most promising to the heuristic (*$k$-best pruning*). The number of promising moves $k$ is subject to empirical optimization. This technique, together with other enhancements such as *killer moves*, has been successfully used for MCTS-IR-M before even in Lines of Action, a domain with an average branching factor twice as high as 6×6 Breakthrough (Winands et al., 2010). Move ordering and $k$-best pruning could make all MCTS-minimax hybrids viable in domains with much higher branching factors, including the newly proposed MCTS-IC-M and MCTS-IP-M. We call the hybrids with activated move ordering and $k$-best pruning *enhanced hybrids* or *MCTS-IR-M-k*, *MCTS-IC-M-k*, and *MCTS-IP-M-k*, respectively.

In contrast to the previous subsection, we only report on the performance of the strongest parameter settings per hybrid and domain in this subsection. Parameter landscapes are not provided. The reason is that all tuning for this subsection was conducted with the help of a



(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, $d = 5$.

Figure 7: Performance of MCTS-IP in Othello, Breakthrough and Catch the Lion.

bandit-based optimizer, distributing test games to parameter settings via the UCB1-TUNED formula. This method speeds up the tuning process as it does not waste samples on clearly inferior settings, but results in win rates with drastically different confidence intervals for different settings.

Another difference to Subsection 5.2 is that we observed overfitting in some cases. Especially MCTS-IP-M-k turned out to be sensitive to the opponent it is tuned against—possibly because it has four parameters as opposed to the three of the other hybrids. Instead of tuning against the comparatively weak baseline MCTS-Solver, all hybrids in this subsection were therefore tuned against a mix of opponents. The mix consists of the best-performing MCTS-IR, MCTS-IC, and MCTS-IP variants (with or without embedded minimax searches) found in Subsection 5.2 for the domain at hand. The reported win rates are results of testing the tuned hybrid in 2000 additional games against a single opponent. The chosen opponent is the overall best-performing player found in Subsection 5.2 for the respective domain, namely MCTS-IP-M in Othello, MCTS-IC-E in Catch the Lion, and MCTS-IP-E in Breakthrough, each with their best-performing settings.

Results are provided for MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k. The techniques MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E are unaffected by the introduction of move ordering and $k$-best pruning, so their performance remains unchanged from Subsection 5.2.

This subsection is organized as follows. Subsection 5.3.1 summarizes the main takeaways. Subsection 5.3.2 explains the move ordering functions used for each game and tests their effectiveness. Next, 5.3.3 to 5.3.5 present experimental results for MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k in all domains. Subsection 5.3.6 tests the hybrids against each other instead of the baseline, confirming the relative strength of MCTS-IP-M-k. The hybrids are then compared across domains in 5.3.7. Afterwards, the effects of different time settings are studied in 5.3.8 and the effects of different branching factors in 5.3.9. Combinations are two hybrids are tested in 5.3.10. Subsection 5.3.11 finally compares the best-performing hybrids to $\alpha\beta$ instead of our MCTS baseline.

### 5.3.1 Main Findings

- MCTS-IR-M-k improves on MCTS-IR-E in all three domains. However, its strength comes from move ordering and $k$-best pruning alone, not from embedded minimax searches: The optimal minimax search depth is the minimal depth of 1.

- MCTS-IC-M-k improves on MCTS-IC-E in Othello and Breakthrough. However, its strength again comes from move ordering and $k$-best pruning, not from minimax – the optimal search depth for minimax is 1 here, too.

- MCTS-IP-M-k improves on MCTS-IP-E (and MCTS-IP-M) in all three domains, and truly profits from the embedded minimax searches (optimal minimax depths are larger than 1 in all test domains). It is significantly stronger than the best player found with unenhanced $\alpha\beta$ in all domains, and stronger than all other hybrids tested.

- A comparison of domains shows that all hybrids are most successful in Catch the Lion, probably due to the higher number of shallow hard traps in this Chess-like domain.

- MCTS-IP-M-k works well at all tested time settings from 250 ms per move to 5 s per move, but can overfit to the time settings it was tuned for.

- MCTS-IP-M-k and MCTS-IC-M-k, the hybrids proposed in this article, become considerably more effective as we increase the branching factor in Breakthrough by enlarging the board from 6×6 to 18×6. MCTS-IR-M-k cannot handle larger branching factors this well due to the much higher number of minimax calls.

- It is potentially useful to combine different ways of using identical domain knowledge in MCTS-minimax hybrids. MCTS-IP-M-k for example profits from the combination with MCTS-IR-M-k in Breakthrough and Catch the Lion.

- The combination of MCTS-IP-M-k and MCTS-IR-M-k outperforms both its MCTS part and its $\alpha\beta$ part in Breakthrough, demonstrating a successful combination of the advantages of the two search approaches. In Catch the Lion and Othello however, regular $\alpha\beta$ is still stronger than the best MCTS hybrids found for each domain.

- In conclusion, MCTS-IP-M-k is the strongest standalone MCTS-minimax hybrid investigated in all three tested domains. The use of enhanced minimax for computing node priors is therefore a promising new technique for integrating domain knowledge into an MCTS framework.

### 5.3.2 Move Ordering Functions

This subsection outlines the heuristic move ordering functions used for each of the three test domains. All move orderings are applied lazily by $\alpha\beta$, meaning that first the highest-ranked move is found and searched, then the second-highest-ranked move is found and searched, etc.—instead of first ranking all available moves and then searching them in that order. When two moves are equally promising to the move ordering function, they are searched in random order.

**Othello.** The move ordering we use for Othello is adapted from the Rolit move ordering described in Nijssen (2013). Moves are ordered according to their location on the board. The heuristic value of a move on each square of the board is shown in Figure 8. Corners, for instance, have the highest heuristic values because discs in the corner are always stable and can provide an anchor for more discs to become stable as well.

**Catch the Lion.** The move ordering we use for Catch the Lion ranks winning moves first. After winning moves it ranks capturing moves, ordered by the value difference between the capturing and the captured piece. Capturing a more valuable piece with a less valuable piece is preferred (see 5.1 for the piece values). This is an idea related to MVV-LVA (*Most Valuable Victim—Least Valuable Aggressor*) capture sorting in computer Chess[1]. After capturing moves, it ranks promotions, and after that all other moves in random order.

**Breakthrough.** For 6×6 Breakthrough, we consider two different move orderings: a weaker one and a stronger one. This allows us to demonstrate the effect that the quality of the move ordering has on the performance of the MCTS-minimax hybrids. The weaker move

[1] https://chessprogramming.wikispaces.com/MVV-LVA

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 4 | 3 | 3 | 4 | 2 | 5 |
| 2 | 1 | 3 | 3 | 3 | 3 | 1 | 2 |
| 4 | 3 | 4 | 4 | 4 | 4 | 3 | 4 |
| 3 | 3 | 4 | 4 | 4 | 4 | 3 | 3 |
| 3 | 3 | 4 | 4 | 4 | 4 | 3 | 3 |
| 4 | 3 | 4 | 4 | 4 | 4 | 3 | 4 |
| 2 | 1 | 3 | 3 | 3 | 3 | 1 | 2 |
| 5 | 2 | 4 | 3 | 3 | 4 | 2 | 5 |

Figure 8: The move ordering for Othello.

ordering ranks moves according to how close they are to the opponent's home row—the closer, the better. The stronger move ordering ranks winning moves first. Second, it ranks saving moves (captures of an opponent piece that is only one move away from winning). Third, it ranks captures, and fourth, all other moves. Within all four groups of moves, moves that are closer to the opponent's home row are preferred as well.

**Effectiveness of the move ordering functions.** The following experiment was conducted in order to test whether the proposed move ordering functions are valid, i.e. whether they rank moves more effectively than a random ordering. Effective move orderings on average lead to quicker cutoffs in $\alpha\beta$ and thus to smaller $\alpha\beta$ trees for a given search depth. For each domain, move ordering, and $\alpha\beta$ search depth from 1 to 5, we therefore played 50 fast games (250 ms per move) between two MCTS-IC-M players, logging the average sizes of $\alpha\beta$ trees for each player. Since MCTS-IC-M uses $\alpha\beta$ in each rollout, these 50 games provided us with a large number of $\alpha\beta$ searches close to typical game trajectories. Tables 1 and 2 presents the results. All move orderings are successful at reducing the average size of $\alpha\beta$ trees in their respective domains. As expected, the stronger move ordering for Breakthrough results in smaller trees than the weaker move ordering. Depth-1 trees are unaffected because no $\alpha\beta$ cutoffs are possible. When comparing the move orderings across domains, it seems that the stronger move ordering in Breakthrough is most effective (88.6% tree size reduction at depth 5, effective branching factor reduced from 7.8 to 5.0), while the Othello move ordering is least effective (29.0% tree size reduction at depth 5, effective branching factor reduced from 4.4 to 4.1). Note that these data do not show whether the gains from move ordering outweigh the overhead, nor whether they allow effective $k$-best pruning—this is tested in the following subsections. We return to time settings of 1 second per move.

### 5.3.3 Experiments with MCTS-IR-M-k

The best-performing settings of MCTS-IR-M-k in the tuning experiments were $d = 1$, $\epsilon = 0$, and $k = 2$ in Othello and Catch the Lion, $d = 1$, $\epsilon = 0$, and $k = 20$ in Breakthrough with the weaker move ordering, and $d = 1$, $\epsilon = 0.1$, and $k = 1$ in Breakthrough with the stronger move ordering. In all domains, no search deeper than one ply proved to be worthwhile ($d = 1$). In Breakthrough with the stronger move ordering, the optimal strategy

| domain | $\alpha\beta$ depth $d$ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Breakthrough | | | | | |
| without m.o. | 17.0 | 90.9 | 657.4 | 2847.9 | 28835.5 |
| weaker m.o. | 16.5 | 60.4 | 365.2 | 1025.0 | 4522.6 |
| stronger m.o. | 16.6 | 46.9 | 258.6 | 751.3 | 3278.6 |
| Othello | | | | | |
| without m.o. | 8.9 | 41.3 | 180.8 | 555.6 | 1605.3 |
| with m.o. | 9.1 | 35.6 | 147.9 | 466.1 | 1139.0 |
| Catch the Lion | | | | | |
| without m.o. | 10.9 | 48.5 | 279.9 | 1183.8 | 5912.9 |
| with m.o. | 10.9 | 30.0 | 128.8 | 427.0 | 1663.6 |

Table 1: Effectiveness of the move orderings (m.o.) in Breakthrough, Othello, and Catch the Lion. The table shows the average size of $\alpha\beta$ trees of depth $d \in \{1 \ldots 5\}$.

| domain | $\alpha\beta$ depth $d$ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Breakthrough | | | | | |
| without m.o. | 17.0 | 9.5 | 8.7 | 7.3 | 7.8 |
| weaker m.o. | 16.5 | 7.8 | 7.1 | 5.7 | 5.4 |
| stronger m.o. | 16.6 | 6.8 | 6.4 | 5.2 | 5.0 |
| Othello | | | | | |
| without m.o. | 8.9 | 6.4 | 5.7 | 4.9 | 4.4 |
| with m.o. | 9.1 | 6.0 | 5.3 | 4.6 | 4.1 |
| Catch the Lion | | | | | |
| without m.o. | 10.9 | 7.0 | 6.5 | 5.9 | 5.7 |
| with m.o. | 10.9 | 5.5 | 5.1 | 4.5 | 4.4 |

Table 2: Effectiveness of the move orderings (m.o.) in Breakthrough, Othello, and Catch the Lion. The table shows the average effective branching factor of $\alpha\beta$ trees of depth $d \in \{1 \ldots 5\}$.

is playing the highest-ordered move in each state without using the evaluation function to differentiate between moves ($k = 1$). Such *move ordering rollouts* (compare Nijssen (2013)) are faster than rollouts that rely on the state evaluation function in our Breakthrough implementation. Each of the best-performing settings played 2000 games against the best player with unenhanced $\alpha\beta$ in the respective domain: MCTS-IP-M in Othello, MCTS-IC-E in Catch the Lion, and MCTS-IP-E in Breakthrough, using the settings found to be optimal in Subsection 5.2. Figure 9 shows the results.

The introduction of move ordering and $k$-best pruning significantly improved the performance of MCTS-IR-M in all domains (p<0.0002) except for Breakthrough with the weaker move ordering. This move ordering turned out not to be effective enough to allow for much pruning, so the problem of the branching factor remained. With the stronger move ordering however, the best-performing setting of MCTS-IP-M-k only considers the highest-ranking move in each rollout step, increasing the win rate against MCTS-IP-E from 49.6% to 77.8%. The stronger move ordering is therefore performing significantly better than the weaker move ordering in MCTS-IR-M-k (p<0.0002).

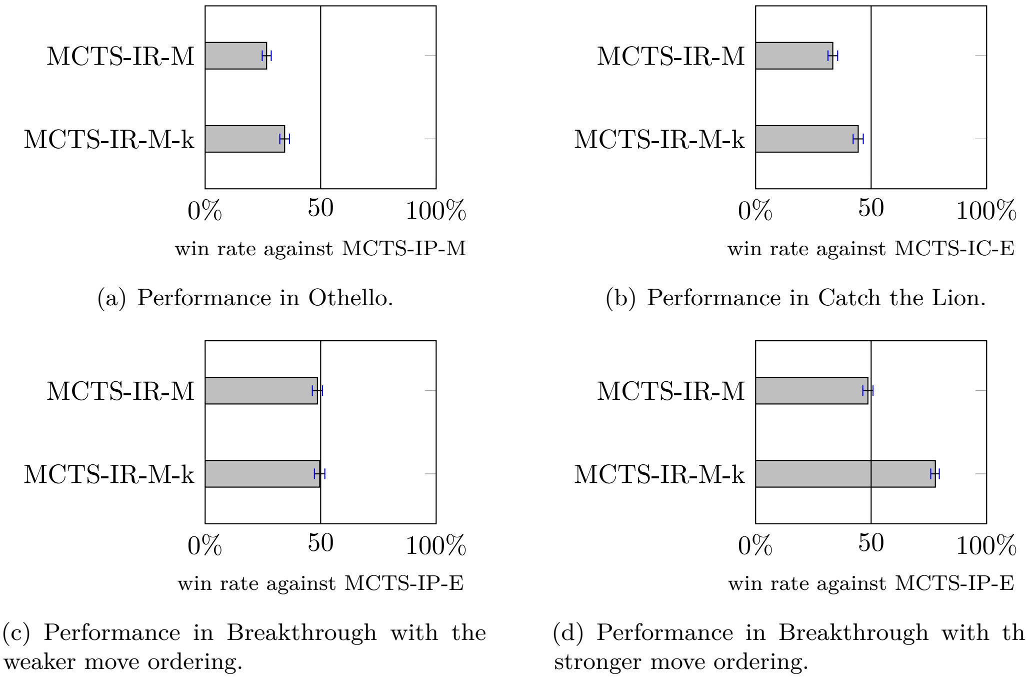
The best-performing settings of MCTS-IR-M-k searched with 830 rps in Othello, 1700 rps in Breakthrough with the weaker move ordering, 13200 rps in Breakthrough with the stronger move ordering, and 32400 rps in Catch the Lion.

As in Subsection 5.2.2, MCTS-IR-M-k was also tested in 2000 games per domain against the non-hybrid, static evaluation version MCTS-IR-E. In this comparison, the enhancements improved the win rate of MCTS-IR-M from 37.1% to 62.8% in Othello, from 35.3% to 80.3% in Breakthrough (strong ordering), and from 47.9% to 65.3% in Catch the Lion. The hybrid version is now significantly stronger than its static equivalent in all three domains. However, the best-performing hybrids do not perform deeper search than MCTS-IR-E—their strength comes from move ordering and $k$-best pruning alone, not from embedded minimax searches. Similar rollout strategies have been successful in Lines of Action before (Winands & Björnsson, 2010).

### 5.3.4 Experiments with MCTS-IC-M-k

The most promising settings of MCTS-IC-M-k in the tuning experiments were $d = 100$, $m = 1$, and $k = 1$ in Othello, $d = 2$, $m = 1$, and $k = 3$ in Catch the Lion, $d = 20$, $m = 5$, and $k = 1$ in Breakthrough with the weaker move ordering, and $d = 15$, $m = 0$, and $k = 1$ in Breakthrough with the stronger move ordering. Note that the settings in Othello and Breakthrough correspond to playing out a large number of moves from the current position, using the best move ordering move in each step, and evaluating the final position. In Othello with $d = 100$, games are played out all the way to terminal positions in this fashion. This makes the best-performing MCTS-IC-M-k players in Othello and Breakthrough similar to the move ordering rollouts discussed in the previous subsection. Only in Catch the Lion, we achieve better play by choosing a small value for $d$ and $k > 1$, leading to an actual embedded minimax search. Move ordering rollouts might still return too unreliable rollout results in this highly tactical domain. Replacing the largest part of rollouts with an evaluation, here through pruned minimax, could be more successful for this reason.

Each of the best-performing settings played 2000 games against MCTS-IP-M in Othello, MCTS-IC-E in Catch the Lion, and MCTS-IP-E in Breakthrough. The results are presented



(a) Performance in Othello.

(b) Performance in Catch the Lion.

(c) Performance in Breakthrough with the weaker move ordering.

(d) Performance in Breakthrough with the stronger move ordering.

Figure 9: Performance of MCTS-IR-M(-k) in Othello, Breakthrough and Catch the Lion. In each game, the algorithm is playing against the best unenhanced player.

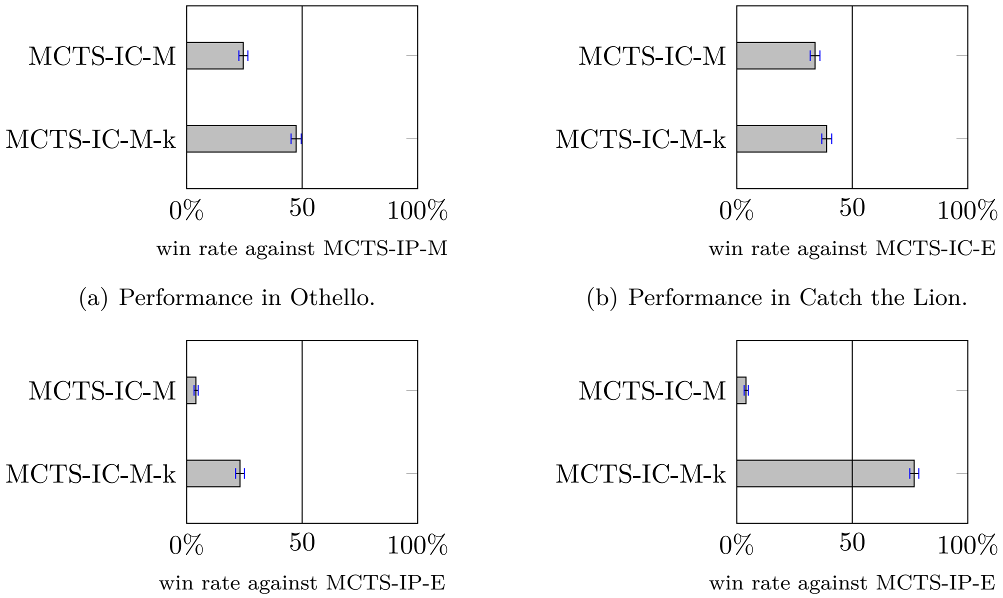
in Figure 10. Move ordering and $k$-best pruning significantly improved the performance of MCTS-IC-M in all domains (p<0.0002 in Othello and Breakthrough, p=0.001 in Catch the Lion). The strength difference between the weaker and the stronger move ordering in Breakthrough is again significant (p<0.0002).

The best-performing settings of MCTS-IC-M-k searched with 7800 rps in Othello, 82600 rps in Breakthrough with the weaker move ordering, 25900 rps in Breakthrough with the stronger move ordering, and 61000 rps in Catch the Lion.

For comparison to Subsection 5.2.3, MCTS-IC-M-k played 2000 games in each domain against MCTS-IC-E. The enhancements improved the win rate of MCTS-IC-M from 26.9% to 66.3% in Othello, from 41.7% to 89.8% in Breakthrough (strong ordering), and from 33.9% to 38.9% in Catch the Lion. The hybrid version is now significantly stronger than its static equivalent in Othello and Breakthrough. However, similar to the case of MCTS-IR-M-k in the previous subsection, the best-performing MCTS-IC-M-k players do not perform true embedded minimax searches with a branching factor of $k > 1$ in these domains. The strength of MCTS-IC-M-k comes from move ordering and $k$-best pruning alone, not from minimax. Only in Catch the Lion, minimax with $k > 1$ is used—but here the simple evaluation function call of MCTS-IC-E still works better.

### 5.3.5 Experiments with MCTS-IP-M-k

The tuning results for MCTS-IP-M-k were $n = 6$, $\gamma = 15000$, $d = 5$, and $k = 10$ in Othello, $n = 4$, $\gamma = 1000$, $d = 7$, and $k = 50$ in Catch the Lion, $n = 3$, $\gamma = 30000$, $d = 5$, and $k = 50$ in Breakthrough with the weaker move ordering, and $n = 2$, $\gamma = 20000$, $d = 8$, and $k = 7$ in



(a) Performance in Othello.

(b) Performance in Catch the Lion.

(c) Performance in Breakthrough with the weaker move ordering.

(d) Performance in Breakthrough with the stronger move ordering.

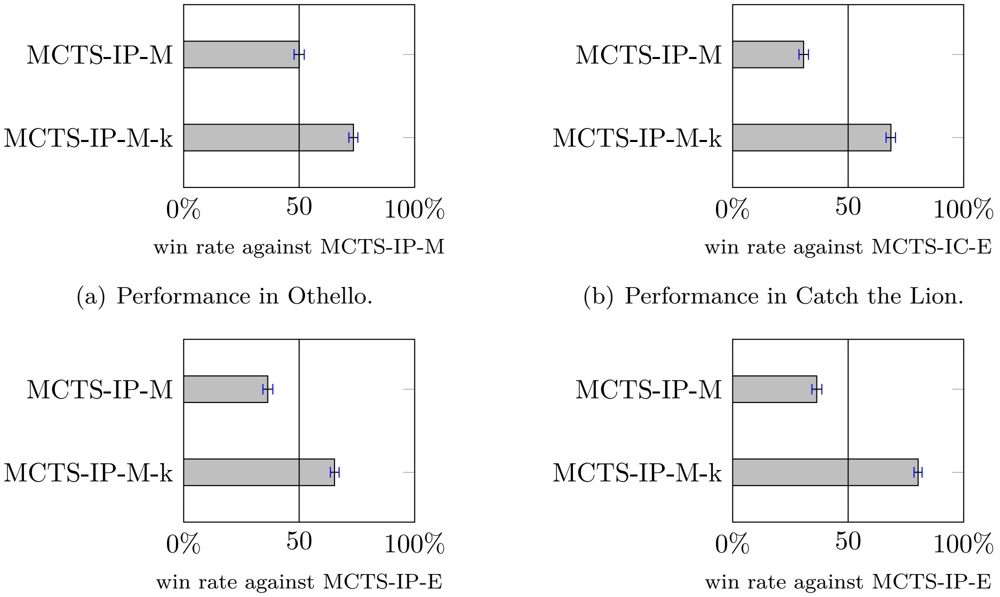
Figure 10: Performance of MCTS-IC-M(-k) in Othello, Breakthrough and Catch the Lion. In each game, the algorithm is playing against the best unenhanced player.

Breakthrough with the stronger move ordering. We can observe that move ordering and $k$-best pruning make it possible to search deeper than in plain MCTS-IP-M. The depth parameter $d$ increases from 3 to 5 in Othello, from 5 to 7 in Catch the Lion, and from 1 to 5 or even 8 in Breakthrough, depending on the strength of the move ordering used. The cost of these deeper searches is further reduced by calling them less often: The visits parameter $n$ increases from 2 to 6 in Othello, from 1 to 4 in Catch the Lion, and from 1 to 2 or 3 in Breakthrough. Thanks to their greater depth, these improved $\alpha\beta$ searches get a stronger weight in all domains except for Catch the Lion. The weight parameter $\gamma$ increases from 5000 to 15000 in Othello, and from 1000 to 20000 or 30000 in Breakthrough. In Catch the Lion, it decreases from 2500 to 1000—a significant strength difference could, however, not be observed.

Each of the settings listed above played 2000 additional games against the best unenhanced player in the respective domain. The results are shown in Figure 11. Move ordering and $k$-best pruning significantly improved the performance of MCTS-IP-M in all domains ($p<0.0002$). Just like with MCTS-IR-M-k and MCTS-IC-M-k, the performance difference between the two Breakthrough move orderings is significant ($p<0.0002$).

For comparison to Subsection 5.2.4, MCTS-IP-M-k was tested in 2000 games per domain against MCTS-IP-E. Move ordering and $k$-best pruning improved the win rate of MCTS-IP-M from 76.2% to 92.4% in Othello, from 36.4% to 80.3% in Breakthrough (strong ordering), and from 97.6% to 98.9% in Catch the Lion. The hybrid version is now significantly stronger than its static equivalent in all three domains.

The best-performing settings of MCTS-IP-M-k searched with 900 rps in Othello, 200 rps in Breakthrough with the weaker move ordering, 50 rps in Breakthrough with the stronger move ordering, and 380 rps in Catch the Lion. In particular in Breakthrough with the stronger move ordering, $\alpha\beta$ search seems to almost completely dominate MCTS with these settings—but as we show in Subsection 5.3.11, MCTS-IP-M-k with an added IR-M-k rollout policy actually outperforms pure $\alpha\beta$ in this domain.



(a) Performance in Othello.

(b) Performance in Catch the Lion.

(c) Performance in Breakthrough with the weaker move ordering.

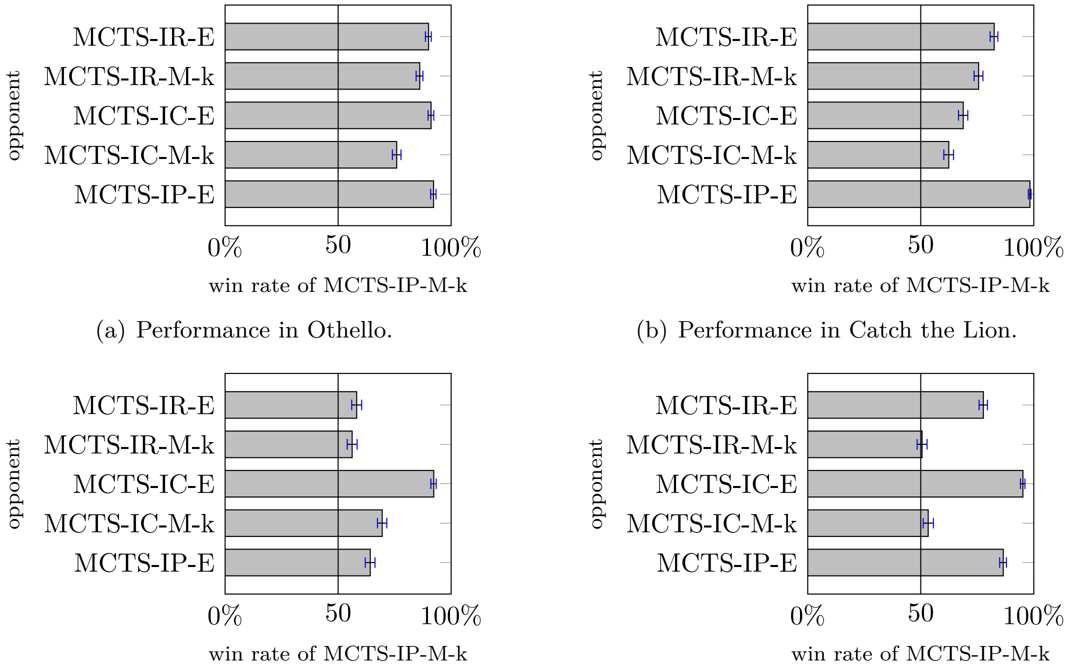
(d) Performance in Breakthrough with the stronger move ordering.

Figure 11: Performance of MCTS-IP-M(-k) in Othello, Breakthrough and Catch the Lion. In each game, the algorithm is playing against the best unenhanced player.

In conclusion, we can say that introducing move ordering and $k$-best pruning has a strong positive effect on all hybrids in all domains. As demonstrated with the example of Breakthrough, the quality of the move ordering used is crucial; but even relatively weak move orderings such as that used in Othello (compare Table 1) are quite effective in improving the MCTS-minimax hybrids. In particular, MCTS-IP-M-k is significantly stronger than the best player found with unenhanced $\alpha\beta$ in all domains (p<0.0002). MCTS-IP-M-k is also the only hybrid that truly profits from embedded minimax searches in all domains, whereas the best-performing variants of MCTS-IR-M-k do not search deeper than one ply, and the strongest versions of MCTS-IC-M-k do not perform real searches in two out of three domains (they reduce the branching factor to 1 and effectively perform move ordering rollouts instead).

### 5.3.6 Comparison of Algorithms

The previous subsections show the performance of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k against a single opponent: the best-performing player without using move ordering and $k$-best pruning. We also tested the best-performing variant of the enhanced hybrids (MCTS-IP-M-k in all domains) against all other discussed algorithms. Each condition consisted of 2000 games. Figure 12 shows the results.



(a) Performance in Othello.

(b) Performance in Catch the Lion.

(c) Performance in Breakthrough with the weaker move ordering.

(d) Performance in Breakthrough with the stronger move ordering.

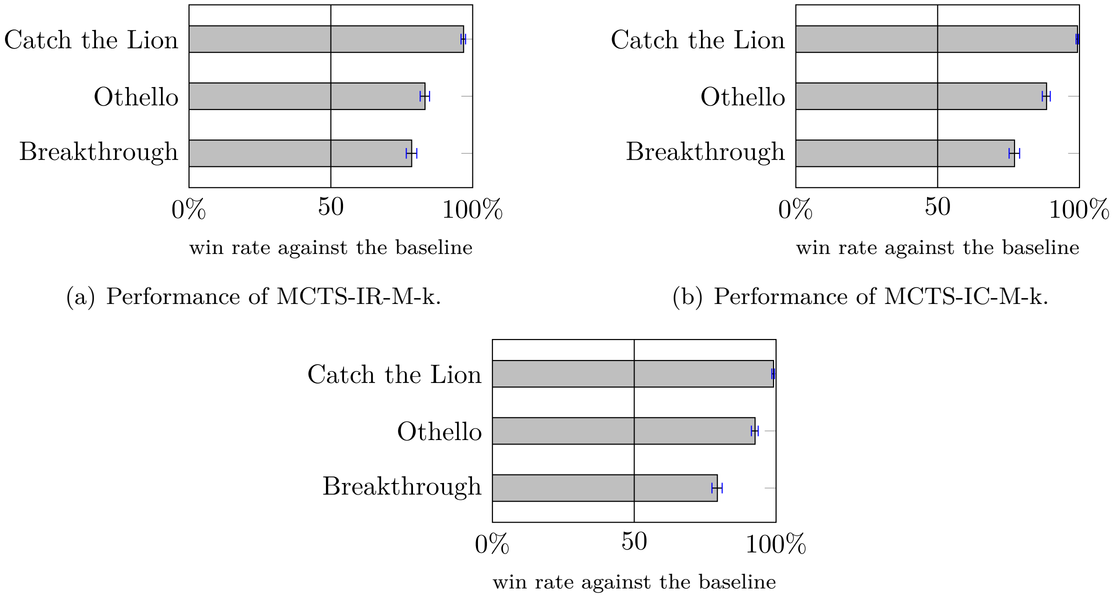
Figure 12: Performance of MCTS-IP-M-k against the other algorithms in Othello, Breakthrough and Catch the Lion.

These results confirm that MCTS-IP-M-k is the strongest standalone MCTS-minimax hybrid tested in this article. It is significantly stronger than MCTS-IR-E, MCTS-IC-E, MCTS-IP-E, MCTS-IR-M-k, and MCTS-IC-M-k in all domains ($p<0.0002$), with the exception of Breakthrough with the stronger move ordering. In this domain it is stronger than MCTS-IC-M-k only with $p<0.05$, and could not be shown to be significantly different in performance from MCTS-IR-M-k. No tested algorithm played significantly better than MCTS-IP-M-k in any domain.

In the remainder of this article, we are always using Breakthrough with the stronger move ordering.

### 5.3.7 Comparison of Domains

In the previous subsections, the experimental results were ordered by domain. Here, we compare the best-performing variants of the enhanced MCTS-minimax hybrids MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k across domains instead, ordered by the type of hybrid. This could help us understand in which games each hybrid might be expected to be most successful. Since Subsections 5.3.3 to 5.3.5 have tested the performance of these hybrids against different opponents—the strongest unenhanced players in each domain—we are testing them against the regular MCTS-Solver baseline here to allow for this cross-domain comparison. Because of the large disparity in playing strength, each condition gave 1000 ms per move to the tested hybrid, and 3000 ms per move to MCTS-Solver. The results are presented in Figure 13.



(a) Performance of MCTS-IR-M-k.

(b) Performance of MCTS-IC-M-k.

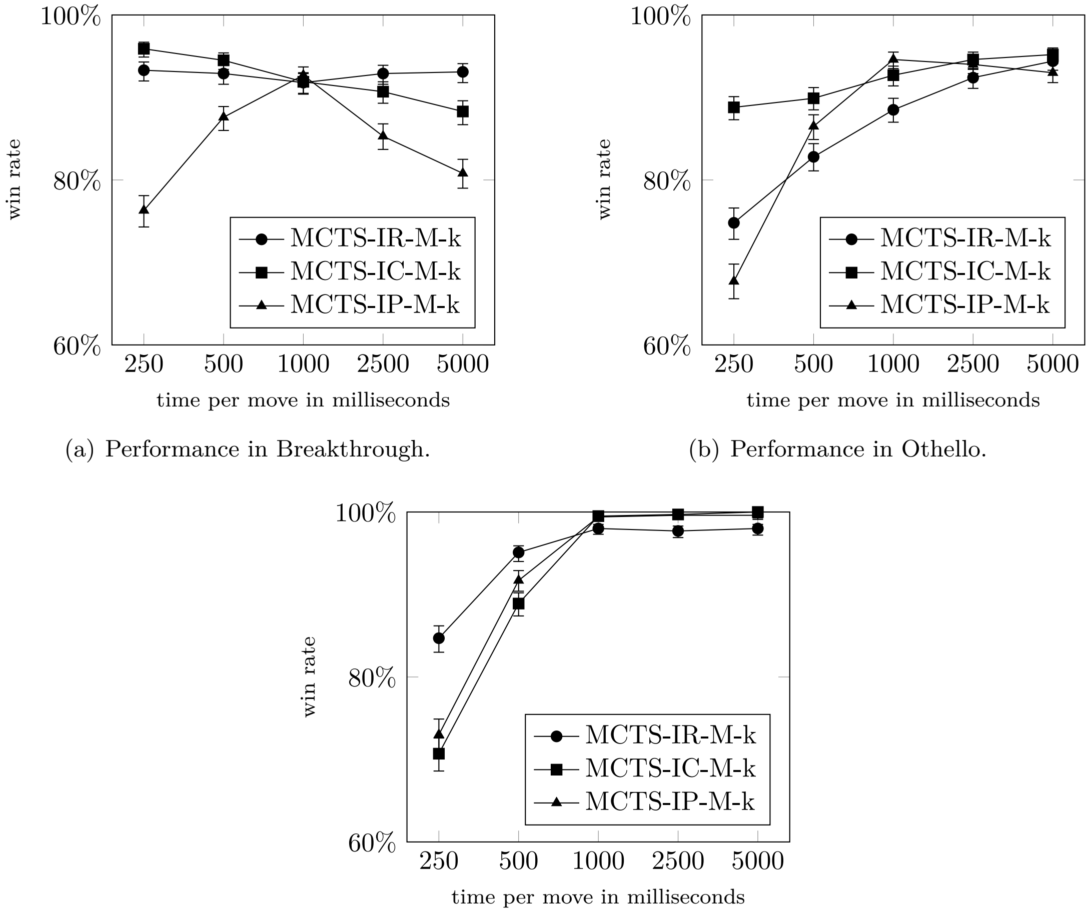
(c) Performance of MCTS-IP-M-k.

Figure 13: Performance of the enhanced MCTS-minimax hybrids across test domains. The best-performing parameter settings are compared for each game. Breakthrough uses the stronger move ordering. The MCTS-Solver baseline uses 300% search time.

Just like their counterparts without move ordering and $k$-best pruning, as well as the knowledge-free hybrids presented in Baier and Winands (2015), all hybrids are most successful in Catch the Lion. It seems that the high number of shallow hard traps in this Chess-like domain makes any kind of embedded minimax searches an effective tool for MCTS, with or without evaluation functions. The hybrids are comparatively least successful in Breakthrough. It remains to be determined by future research whether this is caused by the quality of the move ordering—which still results in a somewhat higher effective branching factor compared to the other domains, as Table 1 shows—, or other factors such as the evaluation function used or features of the search space of Breakthrough.

### 5.3.8 Effect of Time Settings

The results presented in the previous subsections of this article were all based on a time setting of 1000ms per move. In this set of experiments, the best-performing variants of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k played at different time settings from 250 ms per move to 5000 ms per move. The opponent in all conditions was the regular MCTS-Solver baseline at the same time settings as the hybrids, in order to determine how well the hybrids scale with search time. Each condition consisted of 2000 games. Figure 14 shows the results for all test domains.



(a) Performance in Breakthrough.

(b) Performance in Othello.

(c) Performance in Catch the Lion.

Figure 14: Performance of MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k at different time settings. The best-performing parameter settings at 1000 ms are used.

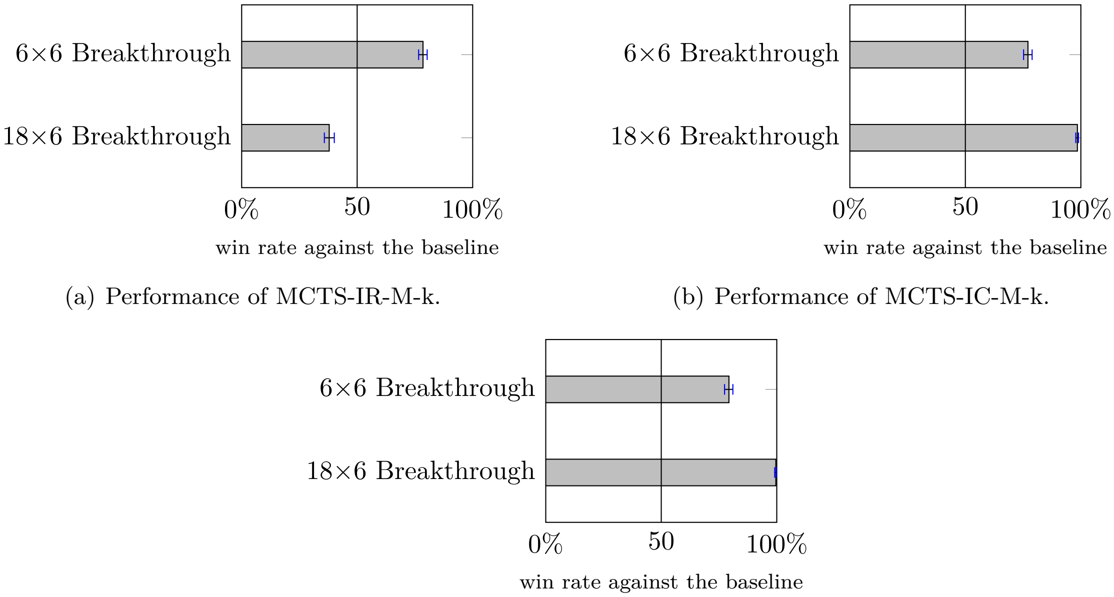
The results indicate that in the range of time settings investigated, it is easier to transfer parameter settings to longer search times than to shorter search times (see in particular Figure 14(c)). MCTS-IP-M-k seems the most likely algorithm to overfit to the specific time setting it was optimized for (see in particular Figure 14(a)). As mentioned before, MCTS-IP-M-k is also most likely to overfit to the algorithm it is tuned against—the flexibility of its four parameters might be the reason. In case playing strength at different time settings is of importance, these time settings could be included in the tuning, just as different opponents were included in the tuning for this article.

Note that these results do not mean the hybrids are ineffective at low time settings. We retuned MCTS-IP-M-k for 250 ms in Breakthrough and Catch the Lion, finding the parameter settings $n = 8$, $\gamma = 30000$, $d = 8$, and $k = 5$ for Breakthrough, and $n = 10$, $\gamma = 10000$, $d = 5$, and $k = 50$ for Catch the Lion. Switching to these variants increased the win rate against the baseline from 76.3% to 96.7% in Breakthrough, and from 73.0% to 90.0% in Catch the Lion. The parameter settings for low time limits are characterized by shortening the embedded minimax searches, either by decreasing $k$ as in Breakthrough or by decreasing $d$ as in Catch the Lion. Increased values of $n$ also mean that the embedded searches are called less often, although from our tuning experience, it seems that the parameter landscape of $n$ and $\gamma$ is fairly flat compared to that of $k$ and $d$ for MCTS-IP-M-k.

### 5.3.9 Effect of Branching Factor

In order to give an indication how well move ordering and $k$-best pruning can help deal with larger branching factors, the enhanced hybrids were also tuned for Breakthrough on an 18×6 board. Increasing the board width from 6 to 18 increases the average branching factor of Breakthrough from 15.5 to 54.2. 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). The best parameter settings found for 18×6 Breakthrough were $d = 1$, $\epsilon = 0.05$, and $k = 1$ for MCTS-IR-M-k, $d = 4$, $m = 0$, and $k = 25$ for MCTS-IC-M-k, and $n = 2$, $\gamma = 20000$, $d = 3$, and $k = 30$ for MCTS-IP-M-k. Figure 15 compares the best-performing settings of the three hybrids on the two board sizes. Each data point represents 2000 games.

The branching factor has a strong effect on hybrids that use minimax in each rollout step: The win rate of MCTS-IR-M-k is reduced from 78.5% (on 6×6) to 37.9% (on 18×6). This is because the computational cost of the minimax searches is much higher on the larger board. While the MCTS-Solver baseline averages about 73600 rollouts per second from the initial 6×6 board position, and MCTS-IR-M-k still achieves about 13200 (17.9%), the speed of the baseline is about 40700 rollouts per second on the 18×6 board, and that of the hybrid only about 2100 (5.2%). The MCTS-minimax hybrids newly proposed in this article, however, show the opposite effect. Both MCTS-IC-M-k and MCTS-IP-M-k become considerably more effective as we increase the branching factor—probably because more legal moves in a given position mean that fewer simulations can be spent on each move, and domain knowledge becomes increasingly more important to obtain useful value estimates. These hybrids do not have to call minimax as often and are therefore not slowed down as much.



(a) Performance of MCTS-IR-M-k.

(b) Performance of MCTS-IC-M-k.

(c) Performance of MCTS-IP-M-k.

Figure 15: Performance of the enhanced MCTS-minimax hybrids in 18×6 Breakthrough. The MCTS-Solver baseline uses 300% search time.

| Domain | MCTS-IP-M-k parameters | | | | MCTS-IR-M-k parameters | | |
|---|---|---|---|---|---|---|---|
| | $n$ | $\gamma$ | $d$ | $k$ | $d$ | $\epsilon$ | $k$ |
| Othello | 6 | 5000 | 5 | 12 | 6 | 0 | 1 |
| Breakthrough | 1 | 500 | 4 | 2 | 1 | 0.05 | 1 |
| Catch the Lion | 10 | 20000 | 7 | 100 | 3 | 0 | 8 |

Table 3: Best-performing parameter settings for MCTS-IP-M-k-IR-M-k.

### 5.3.10 Combination of Hybrids

Subsections 5.3.3 to 5.3.6 show the performance of MCTS-IR-M-k, MCTS-IC-M-k and MCTS-IP-M-k in isolation. In order to get an indication whether the different methods of applying heuristic knowledge can successfully be combined, this subsection provides some results on combinations of different enhanced hybrids. In all three domains, the best-performing hybrid MCTS-IP-M-k as well as MCTS-IC-M-k were combined with MCTS-IR-M-k. The resulting hybrids were named MCTS-IP-M-k-IR-M-k and MCTS-IC-M-k-IR-M-k, respectively. They were tuned against a mix of opponents consisting of the best-performing MCTS-IR, MCTS-IC, and MCTS-IP variants (with or without embedded minimax searches) found in Subsection 5.2, as well as the best-performing MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k variants found in this subsection, for the domain at hand.

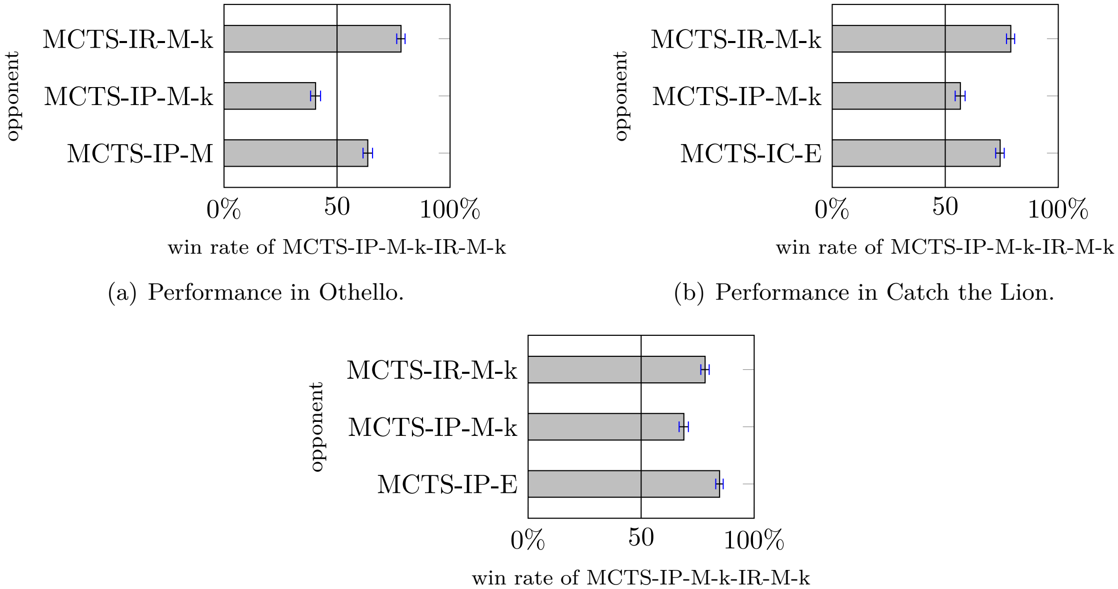
The parameter values found to work best for MCTS-IP-M-k-IR-M-k are presented in Table 3. Tuning of MCTS-IC-M-k-IR-M-k resulted in values of $m = 0$ for Othello and Catch

| Domain | MCTS-IC-M-k parameters | | | MCTS-IR-M-k parameters | | |
|---|---|---|---|---|---|---|
| | $d$ | $m$ | $k$ | $d$ | $\epsilon$ | $k$ |
| Othello | 100 | 0 | 1 | 4 | 0 | 1 |
| Breakthrough | 20 | 1 | 1 | 3 | 0 | 2 |
| Catch the Lion | 7 | 0 | 100 | 2 | 0 | 2 |

Table 4: Best-performing parameter settings for MCTS-IC-M-k-IR-M-k.

the Lion, and $m = 1$ for Breakthrough—meaning that the informed rollout policies are used for at most one move per simulation and have little to no effect on MCTS-IC-M-k. MCTS-IC-M-k-IR-M-k does therefore not seem to be a promising combination of hybrids.

The tuned MCTS-IP-M-k-IR-M-k then played an additional 2000 test games against three opponents: the best unenhanced player in the respective domain in order to have results comparable to Subsections 5.3.3 to 5.3.5, and the two hybrids that the combination consists of (MCTS-IR-M-k and MCTS-IP-M-k). The results are shown in Figure 16.



(a) Performance in Othello.

(b) Performance in Catch the Lion.

(c) Performance in Breakthrough.

Figure 16: Performance of MCTS-IP-M-k combined with MCTS-IR-M-k.

MCTS-IP-M-k-IR-M-k leads to stronger play than the individual hybrids in all three domains, with one exception. In Othello, the combination MCTS-IP-M-k-IR-M-k does perform better than MCTS-IR-M-k, but worse than the stronger component MCTS-IP-M-k. It seems that the slowness of the minimax rollouts outweighs their added strength in this domain.

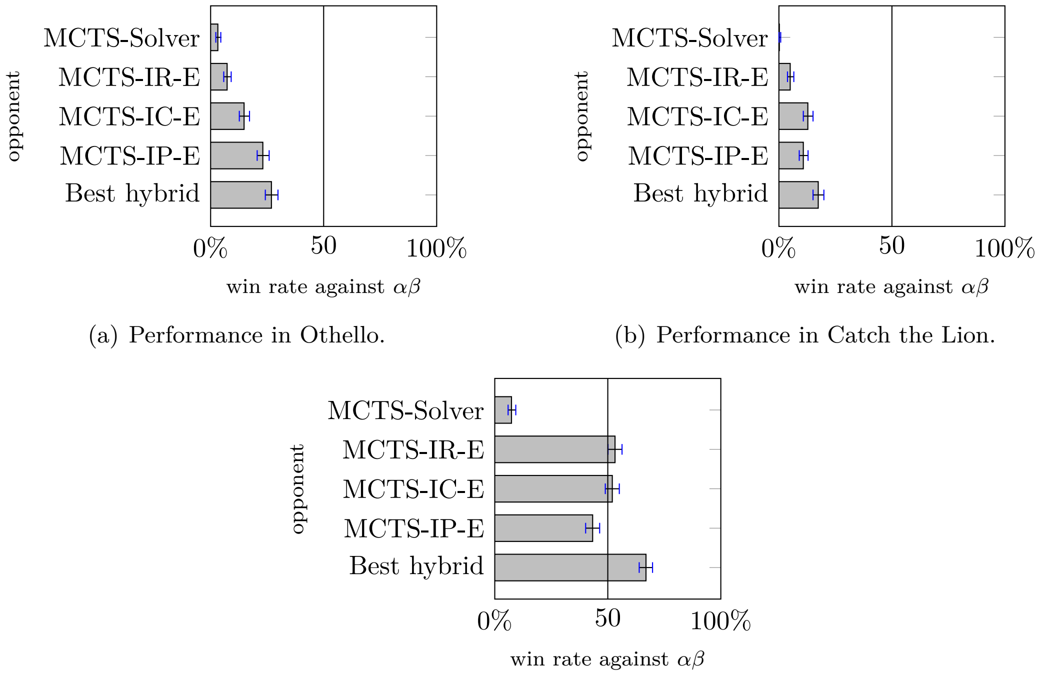
We may conclude it is potentially useful to combine different ways of integrating identical domain knowledge into MCTS-minimax hybrids. Other than MCTS-IC-M-k, MCTS-IP-M-k profits from the combination with MCTS-IR-M-k. When testing MCTS-IP-M-k-IR-M-k and MCTS-IC-M-k-IR-M-k in 2000 games against each other, MCTS-IP-M-k-IR-M-k achieves a win rate of 63.6% in Othello, 74.7% in Breakthrough, and 67.0% in Catch the Lion. This means MCTS-IP-M-k-IR-M-k is significantly stronger than MCTS-IC-M-k-IR-M-k in all domains ($p<0.0002$).

### 5.3.11 Comparison to $\alpha\beta$

Subsections 5.3.3 to 5.3.10 showed the performance of the MCTS-minimax hybrids against various MCTS-based baseline opponents. MCTS-IP-M-k-IR-M-k turned out to be the strongest player tested in Breakthrough and Catch the Lion, and MCTS-IP-M-k with a simple uniformly random rollout policy was the best player tested in Othello. This proves that combining MCTS with minimax can improve on regular MCTS. However, it does not show whether such combinations can also outperform minimax. If the MCTS-minimax hybrids improve on their MCTS component, but not on their minimax component, one could conclude that the test domains are more well-suited to minimax than to MCTS, and that the integration of minimax into MCTS only worked because it resulted in algorithms more similar to minimax. If the MCTS-minimax hybrids can improve on both their MCTS and their minimax component however, they represent a successful combination of the strengths of these two different search methods.

In the last set of experiments in this article, we therefore examined the performance of the strongest hybrid players against minimax. The strongest hybrid players were MCTS-IP-M-k-IR-M-k in Breakthrough and Catch the Lion, and MCTS-IP-M-k in Othello, with the optimal parameter settings found above. In addition, we tested the MCTS-Solver baseline and the best MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E variants in each domain against minimax, in order to get a better picture of how our hybrids compare to state-of-the-art MCTS enhancements. As the minimax baseline we used the same basic $\alpha\beta$ implementation that is also used within the hybrids. It used move ordering and $k$-best pruning as well, with parameter settings tuned against the hybrids. The optimal $k$ for this standalone $\alpha\beta$ was 10 in Breakthrough, 50 in Catch the Lion, and 14 in Othello. In addition, the standalone $\alpha\beta$ used iterative deepening.

1000 games were played for each comparison. The results are shown in Figure 17. The hybrids performed significantly better against $\alpha\beta$ than the MCTS-Solver baseline, MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E, in every domain. The hybrid players achieved 66.9% win rate against $\alpha\beta$ in Breakthrough, 17.4% in Catch the Lion, and 26.9% in Othello. This means that the MCTS-minimax hybrids presented in this article outperform both their MCTS part and their $\alpha\beta$ part in Breakthrough, demonstrating a successful combination of the advantages of the two search approaches. In Catch the Lion and Othello however, regular $\alpha\beta$ is still stronger. This result was expected in the highly tactical domain of Catch the Lion, where the selectivity of MCTS often leads to missing important moves, and the information returned from rollouts seems unreliable (compare with Subsection 5.3.4). The relative strength of $\alpha\beta$ in Othello is more surprising and warrants future investigation. It is possible that the search space of Othello is full of soft traps (Ramanujan et al., 2010b), giving minimax a similar advantage as the hard traps in Catch the Lion.



(a) Performance in Othello.

(b) Performance in Catch the Lion.

(c) Performance in Breakthrough.

Figure 17: Performance against $\alpha\beta$ in Othello, Breakthrough and Catch the Lion.

## 6. Conclusion and Future Research

In this article, we continued the research on MCTS-minimax hybrids for the case where domain knowledge in the form of heuristic evaluation functions is available. Three approaches for integrating such knowledge into MCTS were considered. 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. For all three approaches, we compared the computation of state evaluations through simple evaluation function calls (MCTS-IR-E, MCTS-IC-E, and MCTS-IP-E) to the computation of state evaluations through shallow-depth minimax searches using the same heuristic knowledge (MCTS-IR-M, MCTS-IC-M, and MCTS-IP-M). This hybrid MCTS-minimax technique has only been applied to MCTS-IR before in this form.

Experiments with unenhanced $\alpha\beta$ in the domains of Othello, Breakthrough and Catch the Lion showed that the embedded minimax searches improve MCTS-IP in Othello and Catch the Lion, but are too computationally expensive for MCTS-IR and MCTS-IC in these two domains, as well as for all hybrids in Breakthrough. The main problem of MCTS-minimax hybrids with unenhanced $\alpha\beta$ seems to be the sensitivity to the branching factor of the domain at hand.

Further experiments introduced move ordering and $k$-best pruning to the hybrids in order to cope with this problem, resulting in the enhanced hybrid players called MCTS-IR-M-k, MCTS-IC-M-k, and MCTS-IP-M-k. Results showed that with these relatively simple enhancements, MCTS-IP-M-k is the strongest standalone MCTS-minimax hybrid investigated in this article in all three tested domains. Because MCTS-IP-M-k does not have to call minimax in every rollout or even in every rollout move, it performs better than the other hybrids at low time settings when performance is most sensitive to a reduction in rollouts. MCTS-IP-M-k was also shown to work better for Breakthrough on an enlarged 18×6 board than on the 6×6 board, demonstrating the suitability of the technique for domains with higher branching factors. Additionally, it was shown that the combination of MCTS-IP-M-k with minimax rollouts can lead to further improvements in Breakthrough and Catch the Lion. Moreover, the best-performing hybrid outperformed a simple $\alpha\beta$ implementation in Breakthrough, demonstrating that at least in this domain, MCTS and minimax can be combined to an algorithm stronger than its parts. MCTS-IP-M-k, the use of enhanced minimax for computing node priors, is therefore a promising new technique for integrating domain knowledge into an MCTS framework.

A first direction for future research is the application of additional $\alpha\beta$ enhancements. As a simple static move ordering has proven quite effective in all domains, one could for example experiment with dynamic move ordering techniques such as killer moves or the history heuristic.

Second, some combinations of the hybrids play at a higher level than the hybrids in isolation, despite using the same heuristic knowledge. This may mean we have not yet found a way to fully and optimally exploit this knowledge, which should be investigated further. Combinations of MCTS-IR, MCTS-IC and MCTS-IP could be examined in more detail, as well as new ways of integrating heuristics into MCTS.

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 at equal numbers of rollouts (compare Subsections 5.2.2 to 5.2.4). This phenomenon has only been observed in MCTS-IR before and deserves further study.

Fourth, differences between test domains such as their density of terminal states, their density of hard and soft traps, or their progression property (Finnsson & Björnsson, 2011) could be studied in order to better understand the behavior of MCTS-minimax hybrids with heuristic evaluation functions, and how they compare to standalone MCTS and minimax. The influence of the quality of the evaluation functions and the move ordering functions themselves could also be investigated in greater depth. Comparing Breakthrough with two different move orderings was only a first step in this direction. As mentioned in Baier and Winands (2015), artificial game trees could be a valuable tool to separate the effects of individual domain properties.

## Acknowledgments

This work was supported by the Netherlands Organization for Scientific Research (NWO) in the framework of the project Go4Nature, Grant 612.000.938.

## References

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

Baier, H., & Winands, M. H. M. (2014). Monte-Carlo Tree Search and Minimax Hybrids with Heuristic Evaluation Functions. In Cazenave, T., Winands, M. H. M., & Björnsson, Y. (Eds.), *Computer Games: Third Workshop on Computer Games, CGW 2014*, Vol. 504 of *Communications in Computer and Information Science*, pp. 45–63.

Baier, H., & Winands, M. H. M. (2015). MCTS-Minimax Hybrids. *IEEE Transactions on Computational Intelligence and AI in Games*, *7*(2), 167–179.

Bouzy, B. (2005). Associating Domain-Dependent Knowledge and Monte Carlo Approaches within a Go Program. *Information Sciences*, *175*(4), 247–257.

Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43.

Chaslot, G. M. J.-B., Winands, M. H. M., van den Herik, H. J., Uiterwijk, J. W. H. M., & Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, *4*(3), 343–357.

Coulom, R. (2007). 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*, Vol. 4630 of *Lecture Notes in Computer Science*, pp. 72–83.

Finnsson, H., & Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. In Fox, D., & Gomes, C. P. (Eds.), *23rd AAAI Conference on Artificial Intelligence, AAAI 2008*, pp. 259–264.

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

Gelly, S., & Silver, D. (2007). Combining Online and Offline Knowledge in UCT. In Ghahramani, Z. (Ed.), *24th International Conference on Machine Learning, ICML 2007*, Vol. 227 of *ACM International Conference Proceeding Series*, pp. 273–280.

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Tech. rep., HAL - CCSd - CNRS, France.

Gudmundsson, S. F., & Björnsson, Y. (2013). Sufficiency-Based Selection Strategy for MCTS. In *23rd International Joint Conference on Artificial Intelligence, IJCAI 2013*, pp. 559–565.

Knuth, D. E., & Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, *6*(4), 293–326.

Kocsis, L., & Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In Fürnkranz, J., Scheffer, T., & Spiliopoulou, M. (Eds.), *17th European Conference on Machine Learning, ECML 2006*, Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293.

Lanctot, M., Winands, M. H. M., Pepels, T., & Sturtevant, N. R. (2014). 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.

Lorentz, R. J. (2008). Amazons Discover Monte-Carlo. In van den Herik, H. J., Xu, X., Ma, Z., & Winands, M. H. M. (Eds.), *6th International Conference on Computers and Games, CG 2008*, Vol. 5131 of *Lecture Notes in Computer Science*, pp. 13–24.

Lorentz, R. J. (2011). Experiments with Monte-Carlo Tree Search in the Game of Havannah. *ICGA Journal*, *34*(3), 140–149.

Lorentz, R. J., & Horey, T. (2014). Programming Breakthrough. In van den Herik, H. J., Iida, H., & Plaat, A. (Eds.), *8th International Conference on Computers and Games, CG 2013*, Vol. 8427 of *Lecture Notes in Computer Science*, pp. 49–59.

Nijssen, J. A. M. (2013). *Monte-Carlo Tree Search for Multi-Player Games*. Ph.D. thesis, Maastricht University, Maastricht, The Netherlands.

Nijssen, J. A. M., & Winands, M. H. M. (2012). 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*, Vol. 7168 of *Lecture Notes in Computer Science*, pp. 72–83.

Ramanujan, R., Sabharwal, A., & Selman, B. (2010a). 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.

Ramanujan, R., Sabharwal, A., & Selman, B. (2010b). Understanding Sampling Style Adversarial Search Methods. In Grünwald, P., & Spirtes, P. (Eds.), *26th Conference on Uncertainty in Artificial Intelligence, UAI 2010*, pp. 474–483.

Ramanujan, R., & Selman, B. (2011). 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*, pp. 202–209.

Robilliard, D., Fonlupt, C., & Teytaud, F. (2014). Monte-Carlo Tree Search for the Game of "7 Wonders". In Cazenave, T., Winands, M. H. M., & Björnsson, Y. (Eds.), *Computer Games: Third Workshop on Computer Games, CGW 2014*, Vol. 504 of *Communications in Computer and Information Science*, pp. 64–77.

Rosenbloom, P. S. (1982). A World-Championship-Level Othello Program. *Artificial Intelligence*, *19*(3), 279–320.

Sato, Y., Takahashi, D., & Grimbergen, R. (2010). A Shogi Program Based on Monte-Carlo Tree Search. *ICGA Journal*, *33*(2), 80–92.

Schadd, M. P. D., & Winands, M. H. M. (2011). Best Reply Search for Multiplayer Games. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(1), 57–66.

Sheppard, B. (2002). World-championship-caliber Scrabble. *Artificial Intelligence*, *134*(1-2), 241–275.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den

Driessche, G., Graepel, T., & Hassabis, D. (2017a). Mastering the Game of Go without Human Knowledge. *Nature*, *550*, 354–359.

Silver, D.,  Thomas Hubert, Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., & Hassabis, D. (2017b). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, *abs/1712.01815*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Silver, D., & Tesauro, G. (2009). Monte-Carlo Simulation Balancing. In Danyluk, A. P., Bottou, L., & Littman, M. L. (Eds.), *26th Annual International Conference on Machine Learning, ICML 2009*, Vol. 382 of *ACM International Conference Proceeding Series*, pp. 945–952.

Sturtevant, N. R. (2008). An Analysis of UCT in Multi-Player Games. *ICGA Journal*, *31*(4), 195–208.

Tak, M. J. W., Winands, M. H. M., & Björnsson, Y. (2012). N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(2), 73–83.

Winands, M. H. M., Björnsson, Y., & Saito, J.-T. (2010). Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(4), 239–250.

Winands, M. H. M., & Björnsson, Y. (2010). Evaluation Function Based Monte-Carlo LOA. In van den Herik, H. J., & Spronck, P. (Eds.), *12th International Conference on Advances in Computer Games, ACG 2009*, Vol. 6048 of *Lecture Notes in Computer Science*, pp. 33–44.

Winands, M. H. M., & Björnsson, Y. (2011). 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.

Winands, M. H. M., Björnsson, Y., & Saito, J.-T. (2008). Monte-Carlo Tree Search Solver. In van den Herik, H. J., Xu, X., Ma, Z., & Winands, M. H. M. (Eds.), *6th International Conference on Computers and Games, CG 2008*, Vol. 5131 of *Lecture Notes in Computer Science*, pp. 25–36.