Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups

Marc Lanctot[1], Mark H. M. Winands[1], Tom Pepels[1], and Nathan R. Sturtevant[2]
[1]Games & AI Group, Department of Knowledge Engineering, Maastricht University
[2]Computer Science Department, University of Denver
{marc.lanctot,m.winands,tom.pepels}@maastrichtuniversity.nl, sturtevant@cs.du.edu

***Abstract*—Monte Carlo Tree Search (MCTS) has improved the performance of game engines in domains such as Go, Hex, and general game playing. MCTS has been shown to outperform classic $\alpha\beta$ search in games where good heuristic evaluations are difficult to obtain. In recent years, combining ideas from traditional minimax search in MCTS has been shown to be advantageous in some domains, such as Lines of Action, Amazons, and Breakthrough. In this paper, we propose a new way to use heuristic evaluations to guide the MCTS search by storing the two sources of information, estimated win rates and heuristic evaluations, separately. Rather than using the heuristic evaluations to replace the playouts, our technique backs them up *implicitly* during the MCTS simulations. These minimax values are then used to guide future simulations. We show that using implicit minimax backups leads to stronger play performance in Kalah, Breakthrough, and Lines of Action.**

I. Introduction

Monte Carlo Tree Search (MCTS) [1], [2] is a simulation-based best-first search technique that has been shown to increase performance in domains such as turn-taking games, general-game playing, real-time strategy games, single-agent planning, and more [3]. While the initial applications have been to games where heuristic evaluations are difficult to obtain, progress in MCTS research has shown that heuristics can be effectively be combined in MCTS, even in games where classic minimax search has traditionally been preferred.

The most popular MCTS algorithm is UCT [2], which performs a single simulation from the root of the search tree to a terminal state at each iteration. During the iterative process, a game tree is incrementally built by adding a new leaf node to the tree on each iteration, whose nodes maintain statistical estimates such as average payoffs. With each new simulation, these estimates improve and help to guide future simulations.

In this work, we propose a new technique to augment the quality of MCTS simulations with an implicitly-computed minimax search which uses heuristic evaluations. Unlike previous work, these heuristic evaluations are used as *separate source of information*, and backed up in the same way as in classic minimax search. Furthermore, these minimax-style backups are done *implicitly*, as a simple extra step during the standard updates to the tree nodes, and always maintained separately from win rate estimates obtained from playouts. These two separate information sources are then used to guide MCTS simulations. We show that combining heuristic evaluations in this way can lead to significantly stronger play performance in three separate domains: Kalah, Breakthrough, and Lines of Action.

A. Related Work

Several techniques for minimax-influenced backup rules in the simulation-based MCTS framework have been previously proposed. The first was Coulom's original *maximum backpropagation* [1]. This method of backpropagation suggests, after a number of simulations to a node has been reached, to switch to propagating the maximum value instead of the simulated (average) value. The rationale behind this choice is that after a certain point, the search algorithm should consider a node *converged* and return an estimate of the best value. Maximum backpropagation has also recently been used in other Monte Carlo search algorithms and demonstrated success in probabilistic planning, as an alternative type of forecaster in BRUE [4] and as Bellman backups for online dynamic programming in Trial-based Heuristic Tree Search [5].

The first use of enhancing MCTS using prior knowledge was in Computer Go [6]. In this work, offline-learned knowledge initialized values of expanded nodes increased performance against a significantly strong benchmark player. This technique was also confirmed to be advantageous in Breakthrough [7]. Another way to introduce prior knowledge is via a progressive bias during selection [8], which has significantly increased performance in Go play strength [9].

In games where minimax search performs well, such as Kalah, modifying MCTS to use minimax-style backups and heuristic values instead to replace playouts offers a worthwhile trade-off under different search time settings [10]. Similarly, there is further evidence suggesting not replacing the playout entirely, but terminating them early using heuristic evaluations, has increased the performance in Lines of Action (LOA) [11], Amazons [12], [13], and Breakthrough [7]. In LOA and Amazons, the MCTS players enhanced with evaluation functions outperform their minimax counterparts using the same evaluation function.

One may want to combine minimax backups or searches without using an evaluation function. The prime example is MCTS-Solver [14], which backpropagates proven wins and losses as extra information in MCTS. When a node is proven to be a win or a loss, it no longer needs to be searched. This domain-independent modification greatly enhances MCTS with negligible overhead. Score-bounded MCTS extends this idea to games with multiple outcomes, leading to $\alpha\beta$-style pruning in the tree [15]. One can use shallow-depth minimax searches in the tree to initialize nodes during expansion, enhance the playout, or to help MCTS-Solver in backpropagation [16].

Finally, recent work has attempted to explain and identify some of the shortcomings that arise from estimates in MCTS, specifically compared to situations where classic minimax search has historically performed well [17], [18]. Attempts have been made to overcome the problem of *traps* or *optimistic moves*, *i.e.,* moves that initially seem promising but then later prove to be bad, such as sufficiency thresholds [19] and shallow minimax searches [16].

II. Adversarial Search in Turn-Taking Games

A finite deterministic Markov Decision Process (MDP) is 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. Here, $\mathcal{S}$ is a finite non-empty set of *states*. $\mathcal{A}$ is a finite non-empty set of *actions*, where we denote $\mathcal{A}(s) \subseteq \mathcal{A}$ the set of available actions at state s. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto \Delta\mathcal{S}$ is a *transition function* mapping each state and action to a distribution over successor states. Finally, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto R$ is a *reward function* mapping (state, action, successor state) triplets to numerical rewards.

A two-player perfect information game is an MDP with a specific form. Denote $\mathcal{Z} = \{s \in \mathcal{S} : \mathcal{A}(s) = \emptyset\} \subset \mathcal{S}$ the set of *terminal states*. In addition, for all nonterminal states $s' \in \mathcal{S} - \mathcal{Z}$, $\mathcal{R}(s, a, s') = 0$. There is a *player identity function* $\tau : \mathcal{S} - \mathcal{Z} \mapsto \{1, 2\}$. The rewards $\mathcal{R}(s, a, s')$ are always with respect to the same player and we assume zero-sum games so that rewards with respect to the opponent player are simply negated. In this paper, we assume fully deterministic domains, so $\mathcal{T}(s, a)$ maps s to a single successor state. However, the ideas proposed can be easily extended to domains with stochastic transitions. When it is clear from the context and unless otherwise stated, we denote $s' = \mathcal{T}(s, a)$.

Monte Carlo Tree Search (MCTS) is a simulation-based best-first search algorithm that incrementally builds a tree, $\mathcal{G}$, in memory. Each search starts with from a *root state* $s_0 \in \mathcal{S} - \mathcal{Z}$, and initially sets $\mathcal{G} = \emptyset$. Each simulation samples a trajectory $\rho = (s_0, a_0, s_1, a_1, \cdots, s_n)$, where $s_n \in \mathcal{Z}$ unless the playout is terminated early. The portion of the ρ where $s_i \in \mathcal{G}$ is called the *tree portion* and the remaining portion is called the *playout portion*. In the tree portion, actions are chosen according to some *selection policy*. The first state encountered in the playout portion is *expanded*, added to $\mathcal{G}$. The actions chosen in the playout portion are determined by a specific *playout policy*. States $s \in \mathcal{G}$ are referred to as *nodes* and statistics are maintained for each node s: the cumulative reward, r_s, and visit count, n_s. By popular convention, we define $r_{s,a} = r_{s'}$ where $s' = \mathcal{T}(s, a)$, and similarly $n_{s,a} = n_{s'}$. Also, we use r_s^τ to denote the reward at state s *with respect to player* $\tau(s)$.

Let $\hat{Q}(s, a)$ be an estimator for the value of state-action pair (s, a), where $s \in \mathcal{A}(s)$. One popular estimator is the observed mean $Q(s, a) = r_{s,a}^\tau / n_{s,a}$. The most widely-used selection policy is based on a bandit algorithm called Upper Confidence Bounds (UCB) [20], used in adaptive multistage sampling [21] and in UCT [2], which selects action a' using

$$a' = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} \left\{ \hat{Q}(s, a) + C\sqrt{\frac{\ln n_s}{n_{s,a}}} \right\}, \tag{1}$$

where C is parameter determining the weight of exploration.

III. Implicit Minimax Backups in MCTS

Our proposed technique is based on the following principle: if an evaluation function is available, then it should be possible to augment MCTS to make use of it for a potential gain in performance. Suppose we are given an evaluation function $v_0(s)$ whose range is the same as that of the reward function $\mathcal{R}$. How should MCTS make use of this information? We propose a simple and elegant solution: add another value to maintain at each node, the *implicit minimax evaluation with respect to player* $\tau(s)$, v_s^τ, with $v_{s,a}^\tau$ defined similarly as above. This new value at node s *only* maintains a heuristic minimax value built from the evaluations of subtrees below s. During backpropagation, r_s and n_s are updated in the usual way, and additionally v_s^τ is updated using minimax backup rule based on children values. Then, similarly to RAVE [6], rather than using $\hat{Q} = Q$ for selection in Equation 1, we use

$$\hat{Q}^{IM}(s, a) = (1 - \alpha)\frac{r_{s,a}^\tau}{n_{s,a}} + \alpha v_{s,a}^\tau, \tag{2}$$

where α weights the influence of the heuristic minimax value.

The entire process is summarized in Algorithm 1. There are a few simple additions to standard MCTS, located on lines 2, 8, 13, and 14. During selection, $\hat{Q}^{IM}$ from Equation 2 replaces Q in Equation 1. During backpropagation, the implicit minimax evaluations v_s^τ are updated based on the children's values. For simplicity, a single $\max$ operator is used here since the evaluations are assumed to be in view of player $\tau(s)$. Depending on the implementation, the signs of rewards may depend on $\tau(s)$ and/or $\tau(s')$. For example, a negamax implementation would include sign inversions at the appropriate places to ensure that the payoffs are in view of the current player at each node. Finally, EXPAND adds all children nodes to the tree, sets their implicit minimax values to their initial heuristic values on line 13, and does a one-ply backup on line 14. A more memory-efficient implementation could add just a single child without fundamentally changing the algorithm, as was done in our experiments in Lines of Action.

In essence, this defines a new *information scheme* where each node is augmented with heuristic estimates which are backed-up differently than the Monte Carlo statistics. When MCTS-Solver is enabled, proven values take precedence in the selection policy and the resulting scheme is informative and consistent [22], so Algorithm 1 converges to the optimal choice eventually. However, before a node becomes a proven win or loss, the implicit minimax values act like an heuristic approximation of MCTS-Solver for the portion of the search tree that has not reached terminal states.

IV. Empirical Evaluation

In this section, we thoroughly evaluate the practical performance of the implicit minimax backups technique. Before reporting head-to-head results, we first describe our experimental setup and summarize the techniques that have been used to improve playouts. We then present results on three game domains: Kalah, Breakthrough, and Lines of Action.

Unless otherwise stated, our implementations expand a new node every simulation, the first node encountered that

```
1  SELECT($s$):
2      Let $A'$ be the set of actions $a \in \mathcal{A}(s)$ maximizing
       $\hat{Q}^{IM}(s,a) + C\sqrt{\frac{\ln n_s}{n_{s,a}}}$
3      return $a' \sim$UNIFORM$(A')$
4
5  UPDATE($s, r$):
6      $r_s \leftarrow r_s + r$
7      $n_s \leftarrow n_s + 1$
8      $v^\tau_s \leftarrow \max_{a \in \mathcal{A}(s)} v^\tau_{s,a}$
9
10 SIMULATE($s_{parent}, a_{parent}, s$):
11     if $\exists a \in \mathcal{A}(s), s' = \mathcal{T}(s,a) \notin \mathcal{G}$ then
12         EXPAND($s$)
13         for $a \in \mathcal{A}(s), s' = \mathcal{T}(s,a)$ do $v_{s'} \leftarrow v_0(s')$
14         $v^\tau_s \leftarrow \max_{a \in \mathcal{A}(s)} v^\tau_{s,a}$
15         $r \leftarrow$PLAYOUT($s$)
16         UPDATE($s, r$)
17         return $r$
18     else
19         if $s \in \mathcal{Z}$ then return $\mathcal{R}(s_{parent}, a_{parent}, s)$
20         $a \leftarrow$SELECT($s$)
21         $s' \leftarrow \mathcal{T}(s,a)$
22         $r \leftarrow$SIMULATE($s, a, s'$)
23         UPDATE($s, r$)
24         return $r$
25
26 MCTS($s_0$):
27     while time left do SIMULATE($-, -, s_0$)
28     return $\text{argmax}_{a \in \mathcal{A}(s_0)}\, n_{s_0,a}$
```

Algorithm 1: MCTS with implicit minimax backups.

is not in the tree. MCTS-Solver is enabled in all of the experiments since its overhead is negligible and never decreases performance. After the simulations, the move with the highest visit count is chosen on line 28. Rewards are in $\{-1, 0, 1\}$ representing a loss, draw, and win. Evaluation function values are scaled to $[-1, 1]$ by passing a domain-dependent score differences through a cache-optimized sigmoid function. When simulating, a single game state is modified and moves are undone when returning from the recursive call. Whenever possible, evaluation functions are updated incrementally. All of the experiments include swapped seats to ensure that each player type plays an equal number of games as first player and as second player. All reported win rates are over 1000 played games and search time is set to 1 second unless specifically stated otherwise. Domain-dependent playout policies and optimizations are reported in each subsection.

We compare to and combine our technique with a number of other ones to include domain knowledge. A popular recent technique is *early playout terminations*. When a leaf node of the tree is reached, a fixed-depth early playout termination, hereby abbreviated to "fet$x$", plays x moves according to the playout policy resulting in state s, and then terminates the playout returning $v_0(s)$. This method has shown to improve performance against standard MCTS in Amazons, Kalah, and Breakthrough [7], [10], [13].

A similar technique is *dynamic early terminations*, which periodically checks the evaluation function (or other domain-dependent features) terminating only when some condition is met. This approach has been used as a "mercy rule" in Go [23] and quite successfully in Lines of Action [24]. In our version, which we abbreviate "det$x$", a playout is terminated and returns 1 if $v_0(s) \geq x$ and -1 if $v_0(s) \leq -x$. Another option is to use an ϵ-greedy playout policy that chooses a successor randomly with probability ϵ and successor state with the largest evaluation with probability $1 - \epsilon$, with improved performance in Chinese Checkers [25], [26], abbreviated "egeϵ".

TABLE I: Enhancements tested in Kalah (K), Breakthrough (B), and Lines of Action (L).

Enhancement / Setting	Abbr.	K	B	L
Improved playout policy	ipp		✓	✓
Early playout termination	fet$x$	✓	✓	
Dynamic early termination	det$x$		✓	✓
ϵ-greedy playouts	egeϵ		✓	
Node priors	np		✓	
Maximum backpropagation			✓	
Progressive bias	PB		✓	✓
$\alpha\beta$ playouts				✓
Implicit minimax backups	imα	✓	✓	✓
Simple evaluation function	efRS, efMS	✓	✓	
Sophisticated ev. function	efLH, efWB		✓	✓
Baseline pl. (ege0.1,det0.5)	bl		✓	
Alt. baseline (ipp,fet20,np)	bl'		✓	

To facilitate the discussion, we refer to each enhancement and setting using different labels. These enhancements and labels are described in the text that follows. But, we also include, for reference, a summary of each in Table I.

Experiments are performed in three domains: Kalah, Breakthrough, and Lines of Action. Example images of each game are shown in Appendix A.[1] To tune parameters in Kalah and Breakthrough, hierarchical elimination tournaments are run where each head-to-head match consisted of at least 200 games with seats swapped halfway. Detailed results of these tournaments and comparisons are contained in Appendix B.[1]

A. Kalah

Kalah is a turn-taking game in the Mancala family of games. Each player has six houses, each initially containing four stones, and a store on the endpoint of the board, initially empty. On their turn, a player chooses one of their houses, removes all the stones in it, and "sows" the stones one per house in counter-clockwise fashion, skipping the opponent's store. If the final stone lands in the player's store, that player gets another turn, and there is no limit to the number of consecutive turns taken by same player. If the stone ends on a house owned by the player that contains no stones, then that player captures all the stones in the adjacent opponent house, putting it into the player's store. The game plays until one player's houses are all empty; the opponent then moves their remaining stones to their store. The winner is the player who has collected the most stones in their store. Kalah has been

[1] Appendices are included in [27].

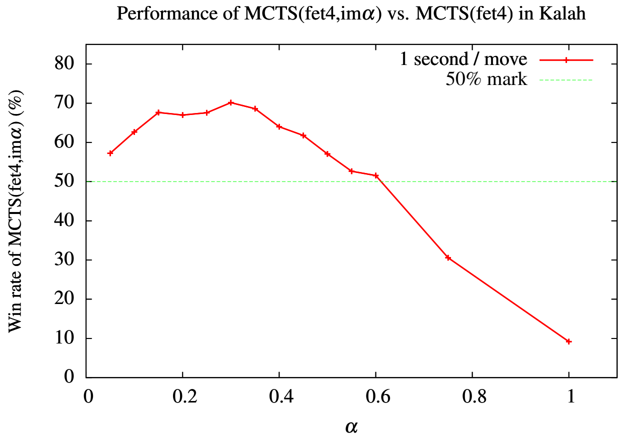

Fig. 1: Results in Kalah. Playouts use fet4. Each data point is based on roughly 1000 games.

weakly solved for several different variants of Kalah [28], and was used as a domain to compare MCTS variants to classic minimax search [10].

In running experiments from the initial position, we observed a noticeable first-player bias. Therefore, as was done in [10], our experiments produce random starting board positions without any stones placed in the stores. Competing players play one game and then swap seats to play a second game using the same board. A player is declared a winner if that player won one of the games and at least tied the other game. If the same side wins both games, the game is discarded.

The default playout policy chooses a move uniformly at random. We determined which playout enhancement led to the best player. Tournament results revealed that a fet4 early termination worked best. The evaluation function was the same one used in [10], the difference between stones in each player's stores. Results with one second of search time are shown in Figure 1. Here, we notice that within the range $\alpha \in [0.1, 0.5]$ there is a clear advantage in performance when using implicit minimax backups against the base player.

B. Breakthrough

Breakthrough is a turn-taking alternating move game played on an 8-by-8 chess board. Each player has 16 identical pieces on their first two rows. A piece is allowed to move forward to an empty square, either straight or diagonal, but may only capture diagonally like Chess pawns. A player wins by moving a single piece to the furthest opponent row.

Breakthrough was first introduced in general game-playing competitions and has been identified as a domain that is particularly difficult for MCTS due to traps and uninformed playouts [19]. Our playout policy always chooses one-ply "decisive" wins and prevents immediate "anti-decisive" losses [29]. Otherwise, a move is selected non-uniformly at random, where capturing undefended pieces are four times more likely than other moves. MCTS with this *improved playout policy* (abbreviated "ipp") beats the one using uniform random 94.3% of the time. This playout policy leads to a clear improvement over random playouts, and so it is enabled by default from this point on.

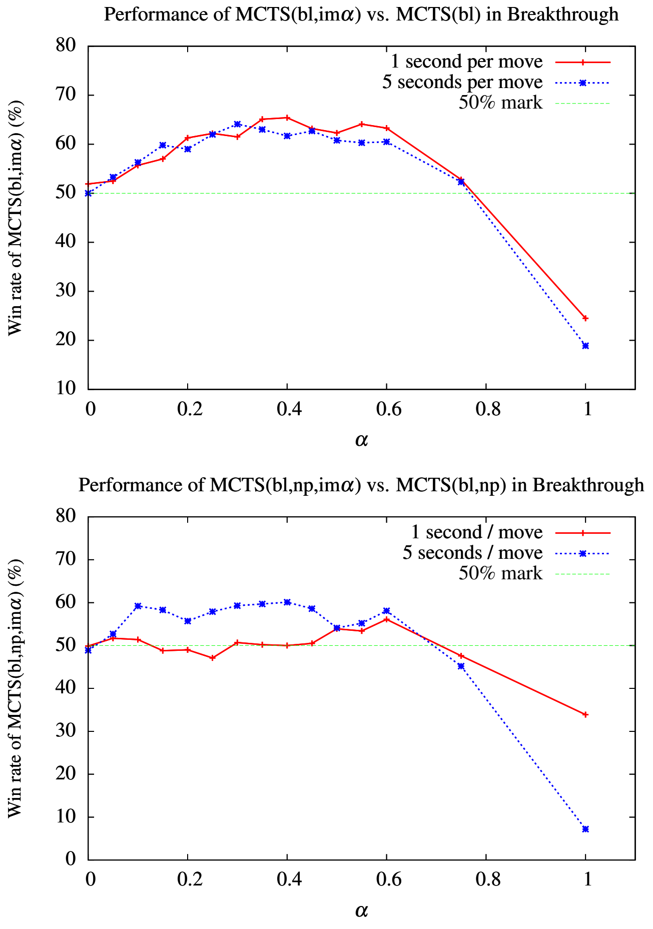

Fig. 2: Results in Breakthrough against baseline player MCTS(ege0.1,det0.5). Each point represents 1000 games. The top graph excludes node priors, bottom graph includes node priors.

In Breakthrough, two different evaluation functions were used. The first one is a simple one found in Maarten Schadd's thesis [30] that assigns each piece a score of 10 and the further row achieved as 2.5, which we abbreviate "efMS". The second one is the more sophisticated one giving specific point values for each individual square per player described in a recent paper by Lorentz & Horey [7], which we abbreviate "efLH". We base much of our analysis in Breakthrough on the Lorentz & Horey player, which at the time of publication had an ELO rating of 1910 on the Little Golem web site.

Our first set of experiments uses the simple evaluation function, efMS. At the end of this subsection, we include experiments for the sophisticated evaluation function efLH.

We first determined the best playout strategy amongst fixed and dynamic early terminations and ϵ-greedy playouts. Our best fixed early terminations player was fet20 and best ϵ-greedy player was ege0.1. Through systematic testing on

Player A	**Player B**	**A Wins** (%)
MCTS(ipp)	MCTS(random playouts)	94.30 ± 1.44
Experiments using only efMS		
MCTS(ege0.1,det0.5)	MCTS(ipp)	68.80 ± 2.88
MCTS(ipp,im0.4)	MCTS(ipp)	82.30 ± 2.37
MCTS(ipp,fet20,im0.4)	MCTS(ipp,fet20)	87.20 ± 2.07
MCTS(bl,im0.4)	MCTS(bl,$\hat{Q}^{CB}$)	67.80 ± 2.90
MCTS(bl,im0.4)	MCTS(bl,$\hat{Q}^{PB}$)	65.50 ± 2.95
MCTS(bl,im0.6)	MCTS(bl)	63.30 ± 2.99
MCTS(bl,im0.6,np)	MCTS(bl)	77.90 ± 2.57
Experiments using efMS and efLH		
MCTS(efMS,bl)	MCTS(efLH,bl')	40.20 ± 3.04
MCTS(efMS,bl,np)	MCTS(efLH,bl')	78.00 ± 2.57
MCTS(efMS,bl,np,im0.4)	MCTS(efLH,bl')	84.90 ± 2.22
MCTS(efMS,bl,im0.4)	MCTS(efLH,bl',im0.6)	53.40 ± 2.19

TABLE II: Summary of results in Breakthrough, with 95% confidence intervals.

	T (in thousands)						
Time	0.1	0.5	1	5	10	20	30
1s	81.9	73.1	69.1	65.2	63.6	66.2	67.0

TABLE III: Win rates (%) of MCTS(bl,im0.4) vs. max backpropagation in Breakthrough, for $T \in \{100, \cdots, 30000\}$.

1000 games per pairing, we determined that the best playout policy when using efMS is the combination (ege0.1,det0.5). The detailed test results are found in Appendix B in [27]. To ensure that this combination of early termination strategies is indeed superior to just the improved playout policy on its own, we also played MCTS(ege0.1,det0.5) against MCTS(ipp). MCTS(ege0.1,det0.5) won 68.8% of these games. MCTS(ege0.1,det0.5) is the best baseline player that we could produce given three separate parameter-tuning tournaments, for all the playout enhancements we have tried using efMS, over thousands of played games. Hence, we use it as our primary benchmark for comparison in the rest of our experiments with efMS. For convenience, we abbreviate this baseline player (MCTS(ege0.1,det0.5)) to MCTS(bl).

We then played MCTS with implicit minimax backups, MCTS(bl,imα), against MCTS(bl) for a variety different values for α. The results are shown in the top of Figure 2. Implicit minimax backups give an advantage for $\alpha \in [0.1, 0.6]$ under both one- and five-second search times. When $\alpha > 0.6$, MCTS(bl,imα) acts like greedy best-first minimax. To verify that the benefit was not only due to the optimized playout policy, we performed two experiments. First, we played MCTS without playout terminations, MCTS(ipp,im0.4) against MCTS(ipp). MCTS(ipp,im0.4) won 82.3% of these games. We then tried giving both players fixed early terminations, and played MCTS(ipp,fet20,im0.4) versus MCTS(ipp,fet20). MCTS(ipp,fet20,im0.4) won 87.2% of these games.

The next question was whether the mixing static evaluation values themselves ($v_0(s)$) at node s was the source of the benefit or whether the minimax backup values (v^τ_s) were the contributing factor. Therefore, we tried MCTS(bl, im0.4) against a baseline player that uses constant bias over the static evaluations, *i.e.,* uses

$$\hat{Q}^{CB}(s,a) = (1-\alpha)Q + \alpha v_0(s'), \text{ where } s' = \mathcal{T}(s,a),$$

and also against a player using a progressive bias of the implicit minimax values, *i.e.,*

$$\hat{Q}^{PB}(s,a) = (1-\alpha)Q + \alpha v^\tau_{s,a}/(n_{s,a}+1),$$

with $\alpha = 0.4$ in both cases. MCTS(bl,im0.4) won 67.8% against MCTS(bl,$\hat{Q}^{CB}$). MCTS(bl,im0.4) won 65.5% against MCTS(bl,$\hat{Q}^{PB}$). A different decay function for the weight placed on v^τ_s could further improve the advantage of implicit minimax backups. We leave this as a topic for future work.

We then evaluated MCTS(im0.4) against *maximum backpropagation* proposed as an alternative backpropagation in the original MCTS work [1]. This enhancement modifies line 24 of the algorithm to the following:

$$\textbf{if } n_s \geq T \textbf{ then return } \max_{a \in \mathcal{A}(s)} \hat{Q}(s,a) \textbf{ else return } r.$$

The results for several values of T are given in Table III.

Another question is whether to prefer implicit minimax backups over *node priors* (abbreviated np) [6], which initializes each new leaf node with wins and losses based on prior knowledge. Node priors were first used in Go, and have also used in path planning problems [31]. We use the scheme that worked well in [7] which takes into account the safety of surrounding pieces, and scales the counts by the time setting (10 for one second, 50 for five seconds). We ran an experiment against the baseline player with node priors enabled, MCTS(bl,imα,np) versus MCTS(bl,np). The results are shown at the bottom of Figure 2. When combined at one second of search time, implicit minimax backups still seem to give an advantage for $\alpha \in [0.5, 0.6]$, and at five seconds gives an advantage for $\alpha \in [0.1, 0.6]$. To verify that the combination is complementary, we played MCTS(bl,im0.6) with and without node priors each against the baseline player. The player with node priors won 77.9% and the one without won 63.3%.

A summary of these comparisons is given in Table II.

MCTS Using Lorentz & Horey Evaluation Function

We now run experiments using the more sophisticated evaluation function from [7], efLH, that assigns specific piece count values depending on their position on the board. Rather than repeating all of the above experiments, we chose simply to compare baselines and to repeat the initial experiment, all using 1 second of search time.

The best playout with this evaluation function is fet20 with node priors, which we call the alternative baseline, abbreviated bl'. That is, we abbreviate MCTS(ipp,fet20,np) to MCTS(bl'). We rerun the initial α experiment using the alternative baseline, which uses the Lorentz & Horey evaluation function, to find the best implicit minimax player using this more sophisticated evaluation function. Results are shown in Figure 3. In this case the best range is $\alpha \in [0.5, 0.6]$ for one second and $\alpha \in [0.5, 0.6]$ for five seconds. We label the best player in this figure using the alternative baseline MCTS(efLH,bl',im0.6).

In an effort to explain the relative strengths of each evaluation function, we then compared the two baseline players. Our baseline MCTS player, MCTS(efMS,bl), wins 40.2% of

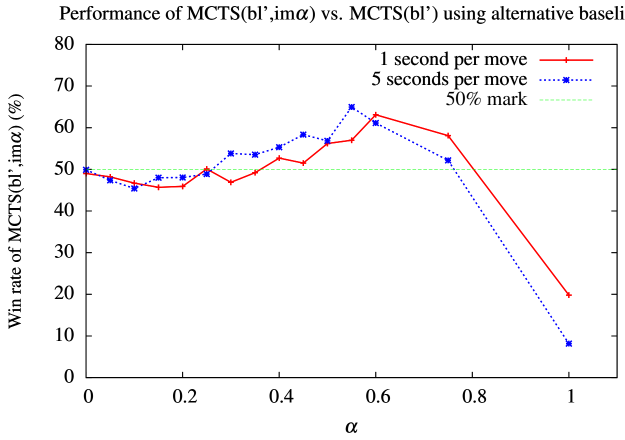

Fig. 3: Results of varying α in Breakthrough using the alternative baseline player. Each point represents 1000 games.

games against the alternative baseline, MCTS(efLH,bl'). When we add node priors, MCTS(efMS,bl,np) wins 78.0% of games against MCTS(efLH,bl'). When we also add implicit minimax backups ($\alpha = 0.4$), the win rate of MCTS(efMS,bl,im0.4,np) versus MCTS(efLH,bl') rises again to 84.9%. Implicit minimax backups improves performance against a stronger benchmark player, even when using a simpler evaluation function.

We then played 2000 games of the two best players for the respective evaluation functions against each other, that is we played MCTS(efMS,bl,np,im0.4) against MCTS(efLH,bl',im0.6). MCTS(efMS,bl,np,im0.4) wins 53.40% of games. Given these results, it could be that a more defensive and less granular evaluation function is preferred in Breakthrough when given only 1 second of search time. The results in our comparison to $\alpha\beta$ in the next subsection seem to suggest this as well.

Comparison to $\alpha\beta$ Search

A natural question is how MCTS with implicit minimax backups compares to $\alpha\beta$ search. So, here we compare MCTS with implicit minimax backups versus $\alpha\beta$ search. Our $\alpha\beta$ search player uses iterative deepening and a static move ordering. The static move ordering is based on the same information used in the improved playout policies: decisive and anti-decisive moves are first, then captures of defenseless pieces, then all other captures, and finally regular moves. The results are listed in Table IV.

The first observation is that the performance of MCTS (vs. $\alpha\beta$) increases as search time increases. This is true in all cases, using either evaluation function, with and without implicit minimax backups. This is similar to observations in Lines of Action [32] and multiplayer MCTS [25], [33].

The second observation is that MCTS(imα) performs significantly better against $\alpha\beta$ than the baseline player at the same search time. Using efMS in Breakthrough with 5 seconds of search time, MCTS(im0.4) performs significantly better than both the baseline MCTS player and $\alpha\beta$ search on their own.

Ev. Func.	Player	Opp.	n	t (s)	Res. (%)
(Both)	$\alpha\beta$(efMS)	$\alpha\beta$(efLH)	2000	1	70.40
(Both)	$\alpha\beta$(efMS)	$\alpha\beta$(efLH)	500	5	53.40
(Both)	$\alpha\beta$(efMS)	$\alpha\beta$(efLH)	400	10	31.25
efMS	MCTS(bl)	$\alpha\beta$	2000	1	27.55
efMS	MCTS(bl)	$\alpha\beta$	1000	5	39.00
efMS	MCTS(bl)	$\alpha\beta$	500	10	47.60
efMS	MCTS(bl,im0.4)	$\alpha\beta$	2000	1	45.05
efMS	MCTS(bl,im0.4)	$\alpha\beta$	1000	5	61.60
efMS	MCTS(bl,im0.4)	$\alpha\beta$	500	10	61.80
efLH	MCTS(bl')	$\alpha\beta$	2000	1	7.90
efLH	MCTS(bl')	$\alpha\beta$	1000	5	10.80
efLH	MCTS(bl')	$\alpha\beta$	500	10	12.60
efLH	MCTS(bl')	$\alpha\beta$	500	20	18.80
efLH	MCTS(bl')	$\alpha\beta$	500	30	19.40
efLH	MCTS(bl')	$\alpha\beta$	500	60	24.95
efLH	MCTS(bl')	$\alpha\beta$	130	120	25.38
efLH	MCTS(bl',im0.6)	$\alpha\beta$	2000	1	28.95
efLH	MCTS(bl',im0.6)	$\alpha\beta$	1000	5	39.30
efLH	MCTS(bl',im0.6)	$\alpha\beta$	500	10	41.20
efLH	MCTS(bl',im0.6)	$\alpha\beta$	500	20	45.80
efLH	MCTS(bl',im0.6)	$\alpha\beta$	500	30	46.20
efLH	MCTS(bl',im0.6)	$\alpha\beta$	500	60	55.60
efLH	MCTS(bl',im0.6)	$\alpha\beta$	130	120	61.54

TABLE IV: Summary of results versus $\alpha\beta$. Here, n represents the number of games played and t time in seconds per search. Win rates are for the Player (in the left column).

The third observation is that MCTS(imα) benefits significantly from weak heuristic information, more so than $\alpha\beta$. When using efMS, MCTS takes less long to do better against $\alpha\beta$, possibly because MCTS makes better use of weaker information. When using efLH, $\alpha\beta$ preforms significantly better against MCTS at low time settings. However, it unclear whether this due to $\alpha\beta$ improving or MCTS worsening. Therefore, we also include a comparison of the $\alpha\beta$ players using efMS versus efLH. What we see is that at 1 second, efMS benefits $\alpha\beta$ more, but as time increases efLH seems to be preferred. Nonetheless, when using efLH, there still seems to be a point where, if given enough search time the performance of MCTS(im0.6) surpasses that of $\alpha\beta$.

C. Lines of Action

In subsection IV-B, we compared the performance of MCTS(imα) to a basic $\alpha\beta$ search player. Our main question at this point is how MCTS(imα) could perform in a game with stronger play due to using proven enhancements in both $\alpha\beta$ and MCTS. For this analysis, we now consider the well-studied game Lines of Action (LOA).

LOA is a turn-taking alternating-move game played on an 8-by-8 board that uses checkers board and pieces. The goal is to connect all your pieces into a single connected group (of any size), where the pieces are connected via adjacent and diagonals squares. A piece may move in any direction, but the number of squares it may move depends on the total number of pieces in the line, including opponent pieces. A piece may jump over its own pieces but not opponent pieces. Captures occur by landing on opponent pieces.

The MCTS player is MC-LOA, whose implementation and enhancements are described in [11]. MC-LOA is a world-champion engine winning the latest Olympiad. The benchmark $\alpha\beta$ player is MIA, the world-best $\alpha\beta$-player upon which MC-LOA is based, winning 4 Olympiads. MC-LOA uses MCTS-Solver, progressive bias, and highly-optimized $\alpha\beta$ playouts.

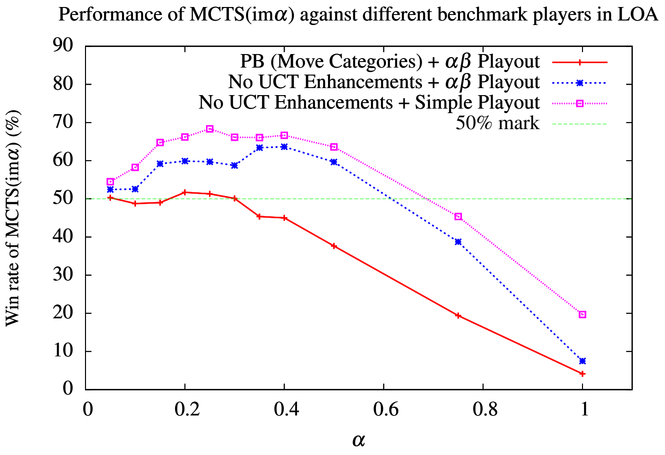

Fig. 4: Results in LOA. Each data point represents 1000 games with 1 second of search time.

Options	Player	Opp.	n	t	Res. (%)
PB	MCTS(imα)	MCTS	32000	1	50.59
PB	MCTS(imα)	MCTS	6000	5	50.91
¬PB	MCTS(imα)	MCTS	1000	1	59.90
¬PB	MCTS(imα)	MCTS	6000	5	63.10
¬PB	MCTS(imα)	MCTS	2600	10	63.80
¬PB	MCTS	$\alpha\beta$	2000	5	40.0
¬PB	MCTS(imα)	$\alpha\beta$	2000	5	51.0
PB	MCTS	$\alpha\beta$	20000	5	61.8
PB	MCTS(imα)	$\alpha\beta$	20000	5	63.3

TABLE V: Summary of results for players and opponent pairings in LOA. All MCTS players use $\alpha\beta$ playouts and MCTS(imα) players use $\alpha = 0.2$. Here, n represents the number of games played and t time in seconds per search.

MIA includes the following enhancements: static move ordering, iterative deepening, killer moves, history heuristic, enhanced transposition table cutoffs, null-move pruning, multi-cut, realization probability search, quiescence search, and negascout/PVS. The evaluation function used is the used in MIA [34]. All of the results in LOA are based 100 opening board positions.[1]

We repeat the implicit minimax backups experiment with varying α. At first, we use standard UCT without enhancements and a simple playout that is selects moves non-uniformly at random based on the move categories, and uses the early cut-off strategy. Then, we enable shallow $\alpha\beta$ searches in the playouts described in [32]. Finally, we enable the progressive bias based on move categories in addition to the $\alpha\beta$ playouts. The results for these three different settings are shown in Figure 4. As before, we notice that in the first two situations, implicit minimax backups with $\alpha \in [0.1, 0.5]$ can lead to better performance. When the progressive bias based on move categories is added, the advantage diminishes. However, we do notice that $\alpha \in [0.05, 0.3]$ seems to not significantly decrease the performance.

Additional results are summarized in Table V. From the graph, we reran $\alpha = 0.2$ with progressive bias for 32000 games giving a statistically significant (95% confidence) win rate of 50.59%. We also tried increasing the search time, in both cases (with and without progressive bias), and observed a gain in performance at five and ten seconds. In the past, the strongest LOA player was MIA, which was based on $\alpha\beta$ search. Therefore, we also test our MCTS with implicit minimax backups against an $\alpha\beta$ player based on MIA. When progressive bias is disabled, implicit minimax backups increases the performance by 11 percentage points. There is also a small increase in performance when progressive bias is enabled. Also, at $\alpha = 0.2$, it seems that there is no statistically significant case of implicit minimax backups hurting performance.

D. *Discussion: Traps and Limitations*

The initial motivation for this work was driven by the trap moves, which pose problems in MCTS [10], [16], [19]. However, in LOA we observed that implicit minimax backups did not speed up MCTS when solving a test set of end game positions. We tried to construct an example board in Breakthrough to demonstrate how implicit minimax backups deals with problems with traps. We were unable to do so. In our experience, traps are effectively handled by the improved playout policy. Even without early terminations, simply having decisive and anti-decisive moves and preferring good capture moves seems to be enough to handle traps in Breakthrough. Also, even with random playouts, an efficient implementation with MCTS-Solver handles shallow traps. Therefore, we believe that the explanation for the advantage offered by implicit minimax backups is more subtle than simply detecting and handling traps. In watching several Breakthrough games, it seems that MCTS with implicit minimax backups builds "fortress" structures [35] that are then handled better than standard MCTS.

While we have shown positive results in a number of domains, we recognize that this technique is not universally applicable. We believe that implicit minimax backups work because there is short-term tactical information, which is not captured in the long-term playouts, but is captured by the implicit minimax procedure. Additionally, we suspect that there must be strategic information in the playouts which is not captured in the shallower minimax backups. Thus, success depends on both the domain and the evaluation function used. We also ran experiments for implicit minimax backups in Chinese Checkers and the card game Hearts, and there was no significant improvement in performance, but more work has to be performed to understand if we would find success with a better evaluation function.

V. Conclusion

We have introduced a new technique called implicit minimax backups for MCTS. This technique stores the information from both sources separately, only combining the two sources to guide selection. Implicit minimax can lead to stronger play even with simple evaluation functions, which are often readily available. In Breakthrough, our evaluation shows that implicit minimax backups increases the strength of MCTS significantly compared to similar techniques for improving

[1] https://dke.maastrichtuniversity.nl/m.winands/loa/

MCTS using domain knowledge. Furthermore, the technique improves performance in LOA, a more complex domain with sophisticated knowledge and strong MCTS and $\alpha\beta$ players. The range $\alpha \in [0.15, 0.4]$ seems to be a safe choice. In Breakthrough, this range is higher, $[0.5, 0.6]$, when using node priors at lower time settings and when using the alternative baseline.

For future work, we would like to apply the technique in other games, such as Amazons, and plan to investigate improving initial evaluations $v_0(s)$ using quiescence search. We hope to compare or combine implicit minimax backups to/with other minimax hybrids from [16]. Differences between $v^{\tau}_{s,a}$ and $Q(s,a)$ could indicate parts of the tree that require more search and hence help guide selection. Parameters could be modified online. For example, α could be changed based on the outcomes of each choice made during the game, and $Q(s,a)$ could be used for online search bootstrapping of evaluation function weights [36]. Finally, the technique could also work in general game-playing using learned evaluation functions [37].

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

References

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *5th International Conference on Computers and Games*, ser. LNCS, vol. 4630, 2007, pp. 72–83.

[2] L. Kocsis and C. Szepesvári, "Bandit-based Monte Carlo planning," in *15th European Conference on Machine Learning*, ser. LNCS, vol. 4212, 2006, pp. 282–293.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. on Comput. Intel. and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[4] Z. Feldman and C. Domshlak, "Monte-Carlo planning: Theoretically fast convergence meets practical efficiency," in *International Conference on Uncertainty in Artificial Intelligence (UAI)*, 2013, pp. 212–221.

[5] T. Keller and M. Helmert, "Trial-based heuristic tree search for finite horizon MDPs," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.

[6] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proceedings of the 24th Annual International Conference on Machine Learning (ICML 2007)*, 2007, pp. 273–280.

[7] R. Lorentz and T. Horey, "Programming Breakthrough," in *8th International Conference on Computers and Games (CG)*, 2013.

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

[9] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding expert knowledge and exploration in Monte-Carlo tree search," in *Advances in Computer Games*, ser. LNCS, vol. 6048, 2010, pp. 1–13.

[10] R. Ramanujan and B. Selman, "Trade-offs in sampling-based adversarial planning," in *21st International Conference on Automated Planning and Scheduling (ICAPS)*, 2011, pp. 202–209.

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

[12] J. Kloetzer, "Monte-Carlo techniques: Applications to the game of Amazons," Ph.D. dissertation, School of Information Science, JAIST, Ishikawa, Japan, 2010.

[13] R. Lorentz, "Amazons discover Monte-Carlo," in *Proceedings of the 6th International Conference on Computers and Games (CG)*, ser. LNCS, vol. 5131, 2008, pp. 13–24.

[14] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte-Carlo tree search solver," in *Computers and Games (CG 2008)*, ser. LNCS, vol. 5131, 2008, pp. 25–36.

[15] T. Cazenave and A. Saffidine, "Score bounded Monte-Carlo tree search," in *International Conference on Computers and Games (CG 2010)*, ser. LNCS, vol. 6515, 2011, pp. 93–104.

[16] H. Baier and M. H. M. Winands, "Monte-Carlo tree search and minimax hybrids," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2013, pp. 129–136.

[17] R. Ramanujan, A. Sabharwal, and B. Selman, "Understanding sampling style adversarial search methods," in *26th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010, pp. 474–483.

[18] ——, "On adversarial search spaces and sampling-based planning," in *20th International Conference on Automated Planning and Scheduling (ICAPS)*, 2010, pp. 242–245.

[19] S. Gudmundsson and Y. Björnsson, "Sufficiency-based selection strategy for MCTS," in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, 2013, pp. 559–565.

[20] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2/3, pp. 235–256, 2002.

[21] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, "An adaptive sampling algorithm for solving Markov Decision Processes," *Operations Research*, vol. 53, no. 1, pp. 126–139, 2005.

[22] A. Saffidine, "Solving games and all that," Ph.D. dissertation, Université Paris-Dauphine, Paris, France, 2013.

[23] B. Bouzy, "Old-fashioned computer Go vs Monte-Carlo Go," in *IEEE Symposium on Computational Intelligence in Games (CIG)*, 2007, Invited Tutorial.

[24] M. H. M. Winands, Y. Björnsson, and J.-T. Saito, "Monte-Carlo tree search solver," in *6th International Conference on Computers and Games (CG 2008)*, ser. LNCS, vol. 5131, 2008, pp. 25–36.

[25] N. R. Sturtevant, "An analysis of UCT in multi-player games," *ICGA Journal*, vol. 31, no. 4, pp. 195–208, 2008.

[26] J. A. M. Nijssen and M. H. M. Winands, "Playout Search for Monte-Carlo Tree Search in Multi-Player Games," in *ACG 2011*, ser. LNCS, vol. 7168, 2012, pp. 72–83.

[27] M. Lanctot, M. H. M. Winands, T. Pepels, and N. R. Surtevant, "Monte Carlo tree search with heuristic evaluations using implicit minimax backups," *CoRR*, vol. abs/1406.0486, 2014, http://arxiv.org/abs/1406.0486.

[28] G. Irving, H. H. L. M. Donkers, and J. W. H. M. Uiterwijk, "Solving Kalah," *ICGA Journal*, vol. 23, no. 3, pp. 139–148, 2000.

[29] F. Teytaud and O. Teytaud, "On the huge benefit of decisive moves in Monte-Carlo tree search algorithms," in *IEEE Conference on Computational Intelligence in Games (CIG)*, 2010, pp. 359–364.

[30] M. P. D. Schadd, "Selective search in games of different complexity," Ph.D. dissertation, Maastricht University, Maastricht, The Netherlands, 2011.

[31] P. Eyerich, T. Keller, and M. Helmert, "High-quality policies for the Canadian travelers problem," in *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence (AAAI)*, 2010, pp. 51–58.

[32] M. H. M. Winands and Y. Björnsson, "$\alpha\beta$-based play-outs in Monte-Carlo tree search," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2011, pp. 110–117.

[33] J. A. M. Nijssen and M. H. M. Winands, "Search policies in multi-player games," *ICGA Journal*, vol. 36, no. 1, pp. 3–21, 2013.

[34] M. H. M. Winands and H. J. van den Herik, "MIA: A world champion LOA program," in *11th Game Programming Workshop in Japan (GPW 2006)*, 2006, pp. 84–91.

[35] M. Guid and I. Bratko, "Detecting fortresses in chess," *Elektrotehniški Vestnik*, vol. 79, no. 1–2, pp. 35–40, 2012.

[36] J. Veness, D. Silver, A. Blair, and W. W. Cohen, "Bootstrapping from game tree search," in *Advances in Neural Information Processing Systems 22*, 2009, pp. 1937–1945.

[37] H. Finnsson and Y. Björnsson, "Learning simulation control in general game playing agents," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010, pp. 954–959.