Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups

Authors

Abstract

Monte Carlo Tree Search (MCTS) has improved the performance of game-playing engines in domains such as Go, Hex, and general-game playing. MCTS has been shown to outperform classic minimax 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 its MCTS simulations. These learned evaluation values are then used to guide future simulations. Compared to current techniques, we show that using implicit minimax backups leads to stronger play performance in Breakthrough, Lines of Action, and Kalah.

Introduction

Monte Carlo Tree Search (MCTS) (Coulom 2007; Kocsis and Szepesvári 2006) is a simulation-based best-first search paradigm 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 (Browne et al. 2012). 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 class minimax search has traditonally been preferred.

The most popular MCTS algorithm is UCT (Kocsis and Szepesvári 2006), 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 track statistical estimates such average payoffs. With each new simulation, these estimates improve and help 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 peformance in three separate domains: Breakthrough, Kalah, and Lines of Action.

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* (Coulom 2007). 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 tree search algorithms and demonstrated success in probabilistic planning, as an alternative type of forecaster in BRUE (Feldman and Domshlak 2013) and as Bellman backups for online dynamic programming in Trial-based Heuristic Tree Search (Keller and Helmert 2013).

The first use of enhancing MCTS using prior knowledge was in Computer Go (Gelly and Silver 2007). In this work, offline-learned knowledge initialized values of expanded nodes increased performance against significantly against strong benchmark player. This technique was also confirmed to be advantageous in Breakthrough (Lorentz and Horey 2013). Another way to introduce prior knowledge is via a progressive bias during selection (Chaslot et al. 2008), which has significantly increased performance in Go play strength (Chaslot et al. 2010).

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 (Ramanujan and Selman 2011a). 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) (Winands, Björnsson, and Saito 2010), Amazons (Kloetzer 2010; Lorentz 2008), and Breakthrough (Lorentz and Horey 2013). 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 (Winands, Björnsson, and Saito 2008a), 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 (Cazenave and Saffidine 2011). Finally, one can use hybrid minimax searches in the tree to initialize nodes during, enhance the playout, or to help MCTS-Solver in backpropagation (Baier and Winands 2013).

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 (Ramanujan, Sabharwal, and Selman 2010b; 2010a). 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 (Gudmundsson and Björnsson 2013) and shallow minimax searches (Baier and Winands 2013).

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, de denote $s' = \mathcal{T}(s, a)$.

Monte Carlo Tree Search 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 $\hat{V}(s)$ an estimator of the win rate starting from node s and $\hat{Q}(s, a)$ for the state-action pair. For example, one popular estimator is the observed mean over all simulations $\bar{Q}(s, a) = r_{s,a}^\tau / n_{s,a}$. The most widely-used selection policy is based on a bandit algorithm called Upper Confidence Bounds (UCB) (Auer, Cesa-Bianchi, and Fischer 2002), used in adaptive multistage sampling (Chang et al. 2005) and in UCT (Kocsis and Szepesvári 2006), which selects action a' using

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

where C is parameter determining the weight of exploration.

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 for MCTS to make use of it, and it should at least never *hurt* performance. But, how could MCTS use this information to account for short-term strategic goals, such as maximizing piece scores?

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? Assuming $v_0(s)$ is a sensible indicator of the reward, we would like that this added source of information strictly benefits MCTS. 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 (Gelly and Silver 2007), rather than using $\hat{Q} = \bar{Q}$ for selection in Equation 1, we use

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

where α is a weight representing the influence of the heuristic minimax values.

The entire process is summarized in Algorithm 1. There are three simple additions to vanilla MCTS, which are located on lines 2, 8, and 13. During selection, $\hat{Q}^{IM}$ from

```
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($s_{prev}, a_{prev}, s$):
11     if $s \notin \mathcal{G}$ then
12         EXPAND($s$)
13         $v^{\tau}_s \leftarrow v^{\tau}_0(s)$
14         $r \leftarrow$PLAYOUT($s$)
15         UPDATE($s, r$)
16         return $r$
17     else
18         if $s \in \mathcal{Z}$ then return $\mathcal{R}(s_{prev}, a_{prev}, s)$
19         $a \leftarrow$SELECT($s$)
20         $s' \leftarrow \mathcal{T}(s,a)$
21         $r \leftarrow$SIMULATE($s, a, s'$)
22         UPDATE($s, r$)
23         return $r$
24
25 MCTS($s_0$):
26     while time left do SIMULATE($-, -, s_0$)
27     return $\text{argmax}_{a \in \mathcal{A}(s_0)} n_{s_0,a}$
```

Algorithm 1: MCTS with implicit minimax backups.

Equation 2 replaces $\bar{Q}$ in Equation 1. During backpropagation, the implicit minimax evaluations v^{τ}_s 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 how the game is modeled, the implementation may require keeping track of or accounting for signs of rewards, for example a negamax model would include a sign switch from the returned child on line 8. Finally, after a node expansion, on line 13, the implicit minimax value is initialized to its heuristic evaluation $v^{\tau}_s \leftarrow v^{\tau}_0(s)$.

In essence, MCTS with implicit minimax backups acts like a heuristic approximation of MCTS-Solver for the portion of the search tree that has not reached terminal states. However, unlike MCTS-Solver and minimax hybrids, these modifications are based on heuristic evaluations rather than proven wins and losses.

Define a binary-outcome game to be one with reward set containing only wins and losses, $R = \{-1, 1\}$. MCTS-Solver changes the backup rule in a way that accounts for proven wins and losses. Define $\top$ as a proven win, $\bot$ as a proven loss ($= +\infty, -\infty$ in MCTS-Solver), we say that backup rule as *minimax-consistent* if $\forall s \in \mathcal{G}$,

$$v^{\tau}_s = \begin{cases} \top & \text{if } \exists a \in \mathcal{A}(s), \text{win}(s,s') \vee (s' \in \mathcal{G} \wedge v^{\tau}_{s'} = \top), \\ \bot & \text{if } \forall a \in \mathcal{A}(s), \text{loss}(s,s') \vee (s' \in \mathcal{G} \wedge v^{\tau}_{s'} = \bot), \end{cases}$$

where $\text{win}(s,s')$ is true if $s' \in \mathcal{Z}$ and leads to a win for player $\tau(s)$ (and false otherwise), and define losee similarly.

Proposition 1. *If Algorithm 1 in run on a binary-outcome game using a minimax-consistent backup rule on line 8 and* $\alpha < 1, C \geq 1$, *then* $\forall s \in \mathcal{S}, \lim_{n_s \to \infty} \hat{V}(s) = V^*(s)$, *where* $V^*(s)$ *is the game-theoretic minimax value of state s.*

For this property to hold for all games and playout policies, the algorithm must continue to explore so that eventually $\mathcal{G} = \mathcal{S}$. On line 2, unvisited actions ($n_{s,a} = 0$) must always be chosen first. The UCT exploration term ensures that each node continues to be visited infinitely often in the limit. The property then follows from the analysis in (Saffidine 2013, Section 2.3). Our technique defines an information scheme that contains both the implicitly computed minimax values as well as win rate estimates, where $\top$ and $\bot$ always override approximations from estimators.

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: Breakthrough, Kalah, and Lines of Action.

Unless otherwise stated, our implementations expand a new node every simulation, the first node encountered that is not in the tree. MCTS-Solver is enabled in all of our experiments since its overhead is negligible and never decreases performance. Rewards are in $\{-1, 0, 1\}$ representing a loss, draw, and win. To ensure values in the same range, evaluation functions are scaled to $[-1, 1]$ by passing a domain-dependant score differences through a cache-optimized sigmoid function. When simulating, to avoid memory overhead, a single game state is modified and moves are undone when returning from the recursive call. Whenever possible, evaluation functions are updated incrementally to save time. 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 unless specifically stated otherwise, such as in Lines of Action. Domain-dependent playout policies and optimizations are reported in each subsection.

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.

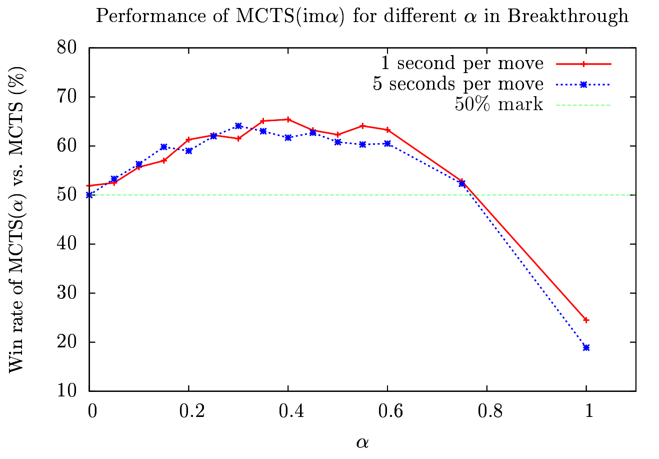
Breakthrough was first introduced in general game-playing competitions and has been identified as a domain that is particularly difficult for MCTS due to traps and uninformed playouts (Gudmundsson and Björnsson 2013). Our playout policy always chooses a one-ply "decisive" wins and prevents immediate "anti-decisive" losses (Teytaud and Teytaud 2010). Otherwise, a move is selected non-uniformly at random, where capturing undefended pieces are four times more likely than other moves. MCTS with this playout policy beats one using uniform random 94.3% of the time. We use two evaluation functions, a simple one found in (Schadd 2011) that assigns each piece a score of 10 and the further row achieved as 2.5, and the one described by (Lorentz and Horey 2013). We base much of our analysis in Breakthrough on the Lorentz & Holey player, which at the time of publication had an ELO rating of 1910 on the Little Golem web site. Currently, the ELO rating is 2010 which is the 8th highest out of over 300 Breakthrough players.

We compare to and combine our technique with number of previous ones to include domain knowledge. A popular technique is *early playout terminations*. When a leaf node of the tree is reached, a fixed-depth early playout terminations, hereby abbreviated to "pd$x$", 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 baseline MCTS in Amazons, Kalah, and Breakthrough (Lorentz 2008; Ramanujan and Selman 2011b; Lorentz and Horey 2013). 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 (Bouzy 2007) and quite successfully in Lines of Action (Winands, Björnsson, and Saito 2008b). 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 (Sturtevant 2008; Nijssen and Winands 2012), abbreviated egeϵ.

We started with a set of experiments using the simple evaluation function. We first determined the best playout strategy amongst fixed and dynamic early terminations and ϵ-greedy playouts. To tune parameters, we ran hierarchical elimination tournaments against players of the same type where each head-to-head match consisted of 200 games with seats swapped halfway. The parameter value sets are given in Appendix A.[1] Our best fixed early terminations player was pd20 and best ϵ-greedy player was ege0.1. Through systematic testing on 1000 games per pairing, we determined that the best playout policy when using the simple evaluation function is the combination (ege0.1,det0.5), which we call the baseline player. The detailed test results are found in Appendix A.

We then played MCTS with implicit minimax backups, MCTS(imα), against this baseline player for a variety different values for α. The results are shown in the top of Figure 1. We see that 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(imα) acts like greedy best-first minimax. To verify that the benefit was not only due to optimized playout policy, we played MCTS(im0.4) against a baseline MCTS player where both used a playout

[1]Appendix A is included as a supplementary material. If accepted, it will be made available as part of a technical report.



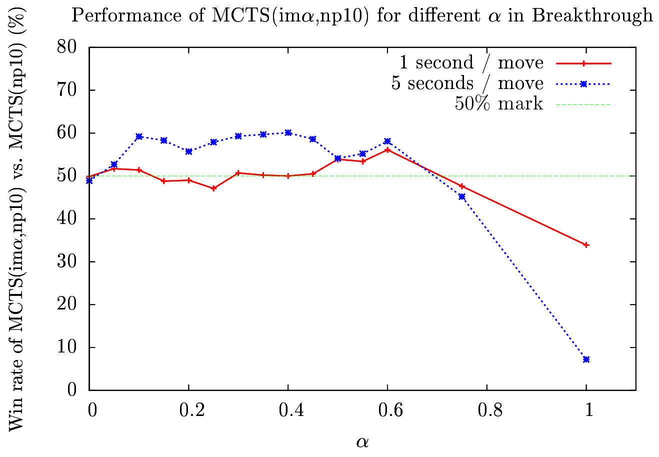


Figure 1: Results in Breakthrough against baseline player MCTS(ege0.1,det0.5). Each point represents 1000 games. Top excludes node priors, bottom includes node priors.

policy without any early terminations, and with both players using pd20. MCTS(im0.4) won 82.3% in the former and 87.2% in the latter.

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_s^τ) were the contributing factor. Therefore, we tried MCTS(im0.4) against a baseline player that uses constant bias over the static evaluations, *i.e.,* uses an estimator $\hat{Q}^{CB}(s,a) = (1-\alpha)\bar{Q} + \alpha v_0(s')$, and also against a player using a progressive bias over the minimax backups, $\hat{Q}^{PB}(s,a) = (1-\alpha)\bar{Q} + \alpha v_s^\tau / (n_{s,a}+1)$. MCTS(im0.4) won 67.8% in the former and 65.5% in the latter. It is possible that a different decay function for v_s^τ will further improve the advantage, and we leave this as an interesting topic for future work.

Another question is whether to prefer implicit minimax over *node priors* (Gelly and Silver 2007), *i.e.,* initializing each new leaf node with wins and losses based on prior knowledge. We use the node priors that worked well in (Lorentz and Horey 2013) which takes into account the

	$T/1000$						
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 1: Win rates (%) of MCTS(im0.4) vs. maximim backpropagation in Breakthrough.

saftey of surrounding pieces, and scaled the counts by the time setting (10 for one second, 50 for five seconds). We ran MCTS(imα) against the baseline player where both players use node priors. The results are shown at the bottom of Figure 1. 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(im0.6) with and without node priors each against the baseline player. The player with node priors won 77.9% and, from Figure 1, the one without won 63.3%.

We then evaluated MCTS(im0.4) against *maximum backpropagation* (Coulom 2007), which modifies line 21 to: if $n_{s,a} \geq T$ return $\max_{a \in \mathcal{A}(s)} \bar{Q}(s,a)$ else return r. The results for several T values are given in Table 1.

Finally, we tried using the more sophisticated evaluation function from (Lorentz and Horey 2013), that assigns specific piece count values depending on their position on the board. Rather than repeat all of the above experiments, we chose simply to compare baselines and to repeat the initial experiment. The best playout with this evaluation function is pd20 with node priors, which we call the alternative baseline. Our baseline MCTS player wins 40.2% of games against the alternative baseline. When we add node priors to our baseline, this rises to 78.0%. When we also add implicit minimax backups ($\alpha = 0.6$), this rises again to 84.9%. We label this best performing player MCTS(im0.6,np). Nonetheless, using the alternative baseline, we reran the initial α experiment. Results are shown in Figure 2. Again, using implicit minimax backups with $\alpha \in [0.1, 0.6]$ gives an advantage in performance. We label the best player in this figure using the alternative baseline MCTS(bl', im0.4, np). We played the two best players against each other, and MCTS(im0.6,np) wins with 76.4%. Given this result, we conjecture that implicit minimax backups may benefit more when combined with defensive and less granular evaluation functions in Breakthrough.

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 then 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

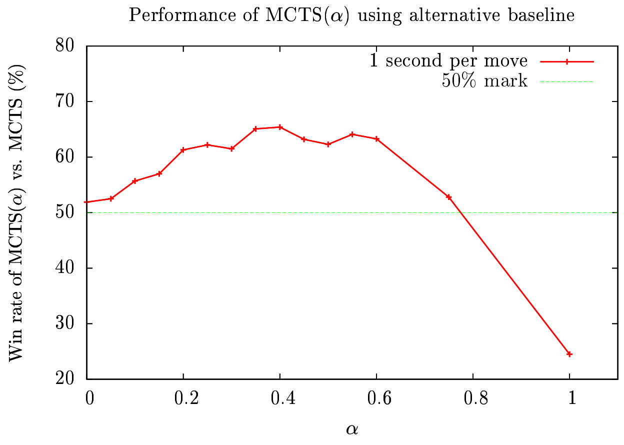


Figure 2: Results of varying α in Breakthrough using the alternative baseline player. Each point represents 1000 games.

■ [ML]$_2$:Note: 5s exps still running...

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 weakly solved for several different variants of Kalah (Irving, Donkers, and Uiterwijk 2000), and was used as a domain to compare MCTS variants to classic minimax search (Ramanujan and Selman 2011a).

In running experiments from the initial position, we observed a noticeable first-player bias. Therefore, as was done in (Ramanujan and Selman 2011a), our experiments produce random starting board positions without any stones placed in the stores. Competiting 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. Tournament results revealed that a pd4 early termination worked best. The evaluation function was simply the difference between stones in each player's stores. Results with one second of search time are shown in Figure 4. Here, we again notice the same patterns as in Breakthrough. 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.

Lines of Action

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.

For LOA, we have based our implementation on the state-of-the-art techniques described in (Winands, Björnsson, and Saito 2010), which has won several Computer Olympiad

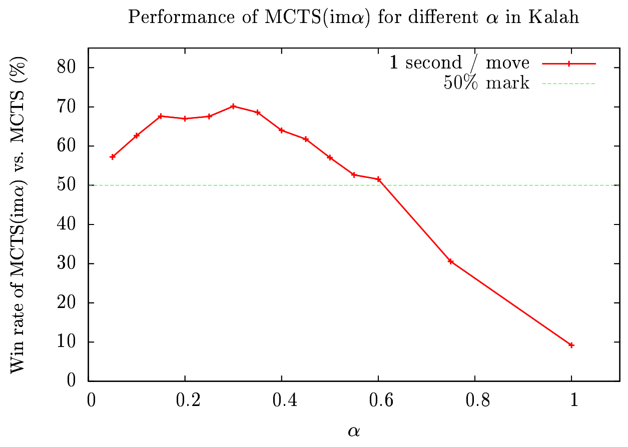


Figure 3: Results in Kalah. Each data point is based on roughly 1000 games; win percentages are $\pm \approx 3.0\%$ with 95% confidence.

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	$\alpha\beta$	4600	5	61.3
PB	MCTS(imα)	$\alpha\beta$	4600	5	62.2

Table 2: 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.

tournaments and is to the best of our knowledge the world's best AI for Lines of Action. The evaluation function used is the one described in (Winands, Björnsson, and Saito 2010) based on the features used in MIA (Winands and van den Herik 2006). All of the results in LOA are based 100 opening board positions.[2]

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 MIA features, and uses the early cut-off strategy. Then, we enable shallow $\alpha\beta$ searches in the playouts described in (Winands and Björnsson 2011). Finally, we enable the progressive bias based on move categories. 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 advantange diminishes. However, we do notice that $\alpha \in [0.05, 0.3]$ seems to

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

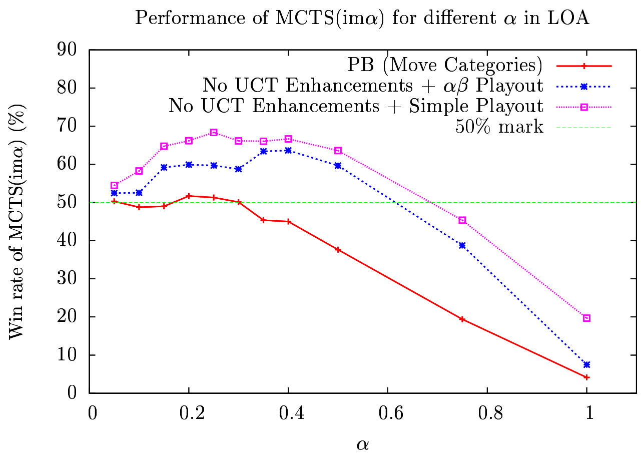


Figure 4: Results in LOA. Each data point represents 1000 games with 1 second of search time.

not significantly decrease the performance.

The results are summarized in Table 2. 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, and observed a gain in performance at five and ten seconds. In the past, the strongest LOA player was MIA which was based on the $\alpha\beta$ search. Therefore, we also test our MCTS with implicit minimax backups against an $\alpha\beta$ player based on MIA. When progress bias is disabled, implicit minimax backups increases the performance by 11 percentage points. There is also notice a small (but statistically insignificant) increase in performance when progressive bias is enabled. Also, at $\alpha = 0.2$, there is no statistically significant case of implicit minimax backups hurting performance.

Conclusion

We have introduced a new technique called implicit minimax backups for MCTS. Unlike previous methods, this techniques stores the information from both sources separately, only combining the two sources to guide selection. This simple technique can lead to stronger play even with basic evaluation functions, in Breakthrough and Kalah. Furthermore, the technique helps performance LOA, a larger, more complex domain with sophisticated knowledge and strong MCTS and classic $\alpha\beta$ players.

For future work, we would like to apply the technique in other games, Amazons in particular. We aim to compare the technique with sufficiency thresholds of Gudmunsson & Björnsson. We also plan to investigate decaying strategies for α and incremental $\alpha\beta$ pruning. The technique could also work in general game-playing agents using evaluations learned during search (Finnsson and Björnsson 2010).

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine*

Learning 47(2/3):235–256.

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

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

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Cazenave, T., and Saffidine, A. 2011. Score bounded Monte-Carlo tree search. In *International Conference on Computers and Games (CG 2010)*, volume 6515 of *LNCS*, 93–104.

Chang, H. S.; Fu, M. C.; Hu, J.; and Marcus, S. I. 2005. An adaptive sampling algorithm for solving Markov Decision Processes. *Operations Research* 53(1):126–139.

Chaslot, G. M. J.-B.; Winands, M. H. M.; Uiterwijk, J. W. H. M.; van den Herik, H. J.; and Bouzy, B. 2008. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation* 4(3):343–357.

Chaslot, G.; Fiter, C.; Hoock, J.-B.; Rimmel, A.; and Teytaud, O. 2010. Adding expert knowledge and exploration in Monte-Carlo tree search. In *Advances in Computer Games*, volume 6048 of *LNCS*, 1–13.

Coulom, R. 2007. Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th International Conference on Computers and Games*, volume 4630 of *LNCS*, 72–83.

Feldman, Z., and Domshlak, C. 2013. Monte-Carlo planning: Theoretically fast convergence meets practical efficiency. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI)*.

Finnsson, H., and Björnsson, Y. 2010. Learning simulation control in general game playing agents. In *Twenty-Fourth AAAI Conference on Articial Intelligence*, 954–959.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *Proceedings of the 24th Annual International Conference on Machine Learning (ICML 2007)*.

Gudmundsson, S., and Björnsson, Y. 2013. Sufficiency-based selection strategy for mcts. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*.

Irving, G.; Donkers, H. H. L. M.; and Uiterwijk, J. W. H. M. 2000. Solving Kalah. *ICGA Journal* 23(3):139–148.

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

Kloetzer, J. 2010. *Monte-Carlo Techniques: Applications to the Game of Amazons*. Ph.D. Dissertation, School of Information Science, Japan Institute of Science and Technology, Ishikawa, Japan.

Kocsis, L., and Szepesvári, C. 2006. Bandit-based Monte Carlo planning. In *15th European Conference on Machine Learning*, volume 4212 of *LNCS*, 282–293.

Lorentz, R., and Horey, T. 2013. Programming breakthrough. In *Proceedings of the 8th International Conference on Computers and Games (CG)*.

Lorentz, R. 2008. Amazons discover Monte-Carlo. In *Proceedings of the 6th International Conference on Computers and Games (CG)*, volume 5131 of *LNCS*, 13–24.

Nijssen, J. A. M., and Winands, M. H. M. 2012. Playout Search for Monte-Carlo Tree Search in Multi-Player Games. In *ACG 2011*, volume 7168 of *LNCS*, 72–83.

Ramanujan, R., and Selman, B. 2011a. Trade-offs in sampling-based adversarial planning. In *13th International Conference on Automated Planning and Scheduling (ICAPS)*, 202–209.

Ramanujan, R., and Selman, B. 2011b. Trade-offs in sampling-based adversarial planning. In *21st International Conference on Automated Planning and Scheduling (ICAPS)*.

Ramanujan, R.; Sabharwal, A.; and Selman, B. 2010a. On adversarial search spaces and sampling-based planning. In *20th International Conference on Automated Planning and Scheduling (ICAPS)*, 242–245.

Ramanujan, R.; Sabharwal, A.; and Selman, B. 2010b. Understanding sampling style adversarial search methods. In *26th Conference on Uncertainty in Artificial Intelligence (UAI)*, 474–483.

Saffidine, A. 2013. *Solving Games and All That*. Ph.D. Dissertation, Universit Paris-Dauphine, Paris, France.

Schadd, M. P. D. 2011. *Selective Search in Games of Different Complexity*. Ph.D. Dissertation, Maastricht University, Maastricht, The Netherlands.

Sturtevant, N. R. 2008. An analysis of UCT in multi-player games. *ICGA Journal* 31(4):195–208.

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

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

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

Winands, M. H. M.; Björnsson, Y.; and Saito, J.-T. 2008a. Monte-Carlo tree search solver. In *Computers and Games (CG 2008)*, volume 5131 of *LNCS*, 25–36.

Winands, M. H. M.; Björnsson, Y.; and Saito, J.-T. 2008b. Monte-Carlo tree search solver. In *6th International Conference on Computers and Games (CG 2008)*, volume 5131 of *LNCS*, 25–36.

Winands, M. H. M.; Björnsson, Y.; and 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.