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 heurisitic 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 model of the game tree is incrementally built by adding a new leaf node to the tree on each iteration. The nodes in the tree are used to store statistical information regarding wins and losses, and backpropagation policies are used to update parent estimates, and these improving estimates are then used to select actions during simulations. When the simulation reaches parts of the game tree not included in the model, a default playout policy is used to simulate to a terminal state where a win or loss is determined.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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. 2008b), which has significantly increased performance in Go play strength (Gelly and Silver 2007).

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). When using MCTS-Solver, proven wins and losses are backpropagated 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 simple, domain-independent modification greatly enhances MCTS, particularly in end-games, 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 2010). 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). We believe that implicit minimax backups help MCTS by taking into account short-term tactical goals such as maximimum piece scores.

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 model, $\mathcal{G}$, of the game tree, in memory. We also refer to $\mathcal{G}$ as the MCTS tree. 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^τ_s 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 $\bar{Q}(s, a) = r^\tau_{s,a}/n_{s,a}$. 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

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

where C is parameter determining the weight of exploration.

Implicit Minimax Backups in MCTS

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^τ_s, with $v^\tau_{s,a}$ 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^τ_s is updated using minimax backup rule based on children values. Then, during selection, rather than using $\hat{Q} = \bar{Q}$ for selection in Equation 1, we use

$$\hat{Q}^{IM}(s, a) = (1 - \alpha)\frac{r^\tau_{s,a}}{n_{s,a}} + \alpha v^\tau_{s,a}. \qquad (2)$$

Pseudo-code is presented 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 Equation 2 replaces $\bar{Q}$ in Equation 1. During backpropagation, the im-

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 $v_s^\tau \leftarrow \max_{a \in \mathcal{A}(s)} v_{s,a}^\tau$
9
10 SIMULATE(s_{prev}, a_{prev}, s):
11 **if** $s \notin \mathcal{G}$ **then**
12 EXPAND(s)
13 $v_s^\tau \leftarrow v_0^\tau(s)$
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** $\operatorname{argmax}_{a \in \mathcal{A}(s_0)} n_{s_0,a}$

Algorithm 1: MCTS with implicit minimax backups.

plicit 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 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. The function $\alpha(n_s)$ will determine how much weight to attribute to these evaluations. Finally, after a node expansion, on line 13, the implicit minimax value is initialized to its heuristic evaluation $v_s^\tau \leftarrow v_0^\tau(s)$.

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, which might help the search in the opening and mid-games.

Define a two-outcome game to be one with reward set containing only wins and losses, $R = \{-1, 1\}$. Define a backup rule as *minimax-consistent* if $\forall s \in \mathcal{G}$,

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

where $\text{win}(s,s')$ is true if $s' \in \mathcal{Z}$ and leads to a win for player $\tau(s)$, and false otherwise, with loss defined similarly. Naturally, we set $\top = +\infty$ and $\bot = -\infty$ which refer to proven wins and losses as computed by MCTS-Solver.

Proposition 1. *If Algorithm 1 in run on a two-outcome game using a minimax-consistent backup rule on line 8, then* $\forall s \in \mathcal{A}, \lim_{n_s \to \infty} \hat{V}(s) = V^*(s)$*, where* $V^*(s)$ *is the unique game-theoretic minimax value for state* s*.*

For this property to hold, the algorithm must continue to explore so that eventually $\mathcal{G} = \mathcal{S}$. Therefore, on line 2 unvisited actions with $n_{s,a} = 0$ must always be chosen before visited ones. The property then follows from the analysis in (Saffidine 2013, Section 2.3). In particular, our technique defines an information scheme that contains both the implicitly computed minimax values as well as win rate estimates.

Of course, while a badly chosen evaluation function with a minimax-consistent backup rule will still converge to the optimal value eventually, we are mainly concerned with how an evaluation function can help MCTS performance. Our empirical evaluation will show that MCTS can benefit significantly from this added information, even with very simple heuristic evaluations.

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 $\tanh$. 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. 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 Table 1. Domain-dependent playout policies and speeds are reported in the appropriate subsections below.

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 sim-

Technique	Parameter set
pd$x$	$\{0, 1, \ldots, 5, 8, 10, 12, 16, 20, 30, 50, 100, 1000\}$
det$x$	$\{.1, .2, .3, , .4, .5, .55, .6, .65, .7, .75, .8, .85, .9\}$
egeϵ	$\{0, .05, .1, .15, .2, .3, .4, .5, .6, .7, .8, .9, 1\}$
imα	$\{0, .05, .1, .15, \ldots, .55, .6, .75, 1\}$

Table 1: Parameter value sets.

ilar 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.

Another popular technique is *maximimum backpropagation*, where certain number of visits in the UPDATE procedure, the valued backed up is the one of the best child instead of the sampled value from the simulation. Couloum originally showed that the the mean backpropagation to be superior in Go (Coulom 2007). In Kalah, the UCTMAX$_H$ algorithm proposed by Ramanujan resembles closely maximum backpropagation with early terminations (pd0) except that the values are also weighted by visit counts, and was shown to perform as well or better than a strong minimax player at a low number of nodes expansions. Another successful technique in Go is *node priors* (Gelly and Silver 2007), where newly-expanded nodes are assigned some initial wins and loss counts, which also improved performance in Breakthrough (Lorentz and Horey 2013). Finally, another way to use heuristic knowledge is as a means of *progressive bias* which has been rather successful in Go (Gelly and Silver 2007; Chaslot et al. 2008a) and Lines of Action (Winands, Björnsson, and Saito 2010).

Breakthrough

Breakthrough is a turn-taking alternating move game played on an 8-by-8 chess board, where each player contains 16 identical pieces on their initial two rows. Each piece is allowed to move forward to an empty square, either straight or via the diagonals, but may only capture via the diagonals like Chess pawns. The goal of the game is bring a 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 random playouts (Gudmundsson and Björnsson 2013). Our playout policy always choose a 1-ply "decisive" move and prevent 1-ply "anti-decisive" losses. Otherwise, a move is selected non-uniformly at random, where capturing undefended pieces are four times more likely than other moves. We use two evaluation functions, a simple one found in

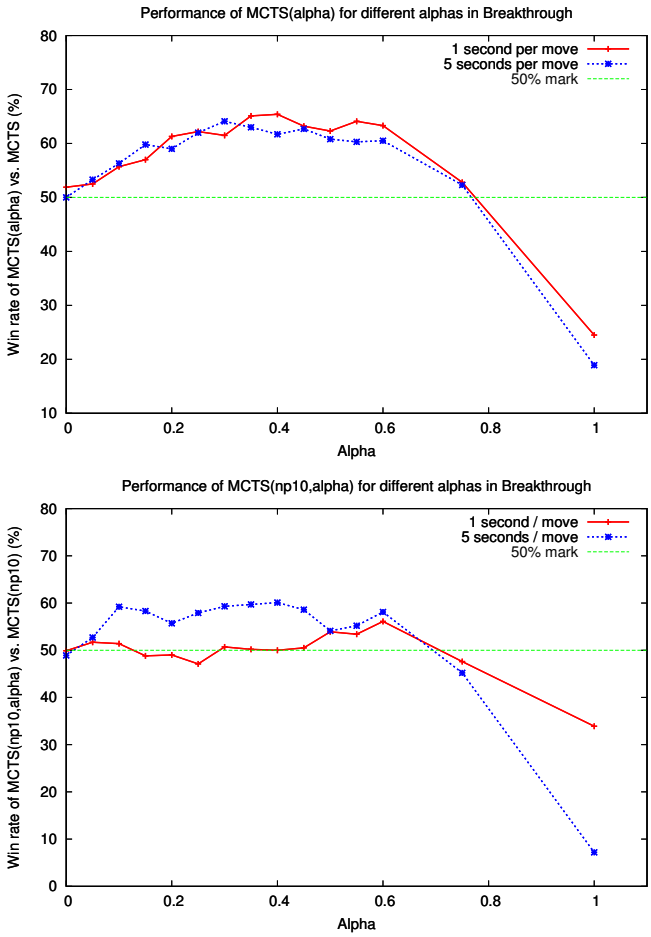

Figure 1: Results in Breakthrough against baseline player MCTS(ege0.1,det0.5). Each point represents 1000 games. Top excludes node priors, bottom includes node priors.

(Schadd 2011) that assigns each piece a score of 10 and the further row achieved as 2.5, and the one found in (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 and is the 8th highest-rated player out of over 300 Breakthrough players.

We first determined the best playout strategy amongst fixed and dynamic early terminations and ϵ-greedy playouts. Through systematic testing on thousands of games, we determined that the best playout policy when using the simple evaluation function is the combination (ege0.1,det0.5). This was suprising since fixed early terminations are much faster, however is likely explained by less overhead due to incremental evaluations and efficient transitions.

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

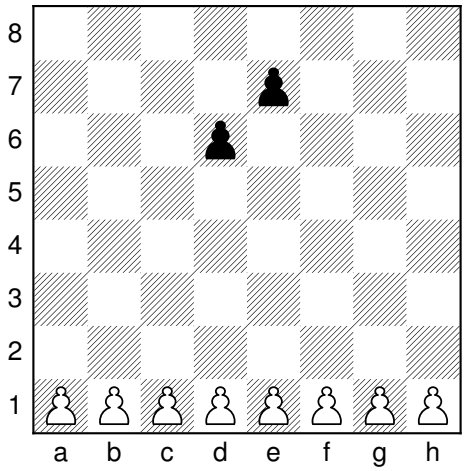

Figure 2: An example position in Breakthrough. ■ [ML]$_2$:I hope to have a nice example to show here soon.

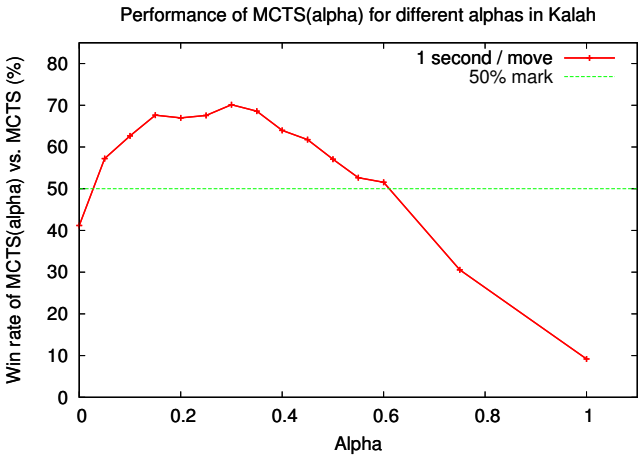

Figure 3: Results in Kalah. Each data point is $\pm$ 3% at the moment (still running), with 1 second of search time. Same evaluation function as (Ramanujan and Selman 2011a) normalized to $[0, 1]$ using tanh and an (optimal) pdepth of 4.

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 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 classical minimax search (Ramanujan and Selman 2011a).

Lines of Action

Lines of Action is a turn-taking alternating-move game played on an 8-by-8 board that uses checkers board and pieces, invented by Claude Soucie around 1960 and published in Sid Sackson's *A Gamut of Games*. 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.

Conclusion

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 *Proceedings of the 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. 2010. Score bounded Monte Carlo Tree Search. In van den Herik, H.; Iida, H.; and Plaat, A., eds., *Proceedings of the 7th International Conference on Computers and Games (CG 2010)*, volume 6515 of *LNCS*. Kanazawa, Japan: Springer. 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.; Winands, M. H. M.; van den Herik, H. J.; Uiterwijk, J.; and Bouzy, B. 2008a. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation* 4(3):343–357.

Chaslot, G. M. J.-B.; Winands, M. H. M.; Uiterwijk, J. W. H. M.; van den Herik, H. J.; and Bouzy, B. 2008b. 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 *Proceedings of the 5th International Conference on Computers and Games (CG'06)*, volume 4630 of *LNCS*, 72–83. Berlin, Heidelberg: Springer-Verlag.

Feldman, Z., and Domshlak, C. 2013. Monte-Carlo plan-

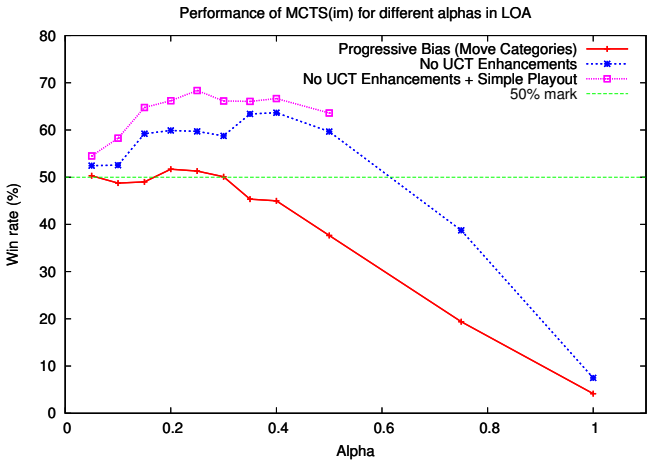

Figure 4: Results in LOA. Each data point represents 1000 games with 5 seconds of search time.

ning: Theoretically fast convergence meets practical efficiency. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI)*.

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 *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*.

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 van den Herik, H. J., and Plaat, A., eds., *Advances in Computer Games (ACG 2011)*, volume 7168 of *LNCS*, 72–83. Berlin, Germany: Springer-Verlag.

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

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

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

Ramanujan, R.; Sabharwal, A.; and Selman, B. 2010b. Understanding sampling style adversarial search methods. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, 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. *International Computer Games Journal* 31(4):195–208.

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. Springer, Berlin Heidelberg.

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

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.