ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/286733609

Enhancing upper confidence bounds for trees with temporal difference values

Conference Paper · August 2014

DOI: 10.1109/CIG.2014.6932895

CITATIONS
3

READS
124

2 authors:

Tom Vodopivec
University of Ljubljana
7 PUBLICATIONS **21** CITATIONS
SEE PROFILE

Branko Šter
University of Ljubljana
41 PUBLICATIONS **303** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Örenäs Research Group View project

All content following this page was uploaded by Branko Šter on 03 April 2018.

The user has requested enhancement of the downloaded file.

Enhancing Upper Confidence Bounds for Trees with Temporal Difference Values

Tom Vodopivec and Branko Šter
University of Ljubljana, Faculty of Computer and Information Science
Tržaška cesta 25, 1000 Ljubljana, Slovenia
{tom.vodopivec, branko.ster}@fri.uni-lj.si

***Abstract*—Upper confidence bounds for trees (UCT) is one of the most popular and generally effective Monte Carlo tree search (MCTS) algorithms. However, in practice it is relatively weak when not aided by additional enhancements. Improving its performance without reducing generality is a current research challenge. We introduce a new domain-independent UCT enhancement based on the theory of reinforcement learning. Our approach estimates state values in the UCT tree by employing temporal difference (TD) learning, which is known to outperform plain Monte Carlo sampling in certain domains. We present three adaptations of the TD(λ) algorithm to the UCT's tree policy and backpropagation step. Evaluations on four games (Gomoku, Hex, Connect Four, and Tic Tac Toe) reveal that our approach increases UCT's level of play comparably to the rapid action value estimation (RAVE) enhancement. Furthermore, it proves highly compatible with a modified all moves as first heuristic, where it considerably outperforms RAVE. The findings suggest that integration of TD learning into MCTS deserves further research, which may form a new class of MCTS enhancements.**

I. Introduction

Over the last few years *Monte Carlo tree search* (MCTS) methods [1] established themselves as the state-of-the-art choice for game playing algorithms [2]. The most notable success was achieving human master level in the complex game of Go [3], which was thought of being decades away. The field gained additional renown with top achievements in international game tournaments – e.g., at General Game Playing [4], Hex [5], Amazons [6] and other games.

Many MCTS-based players use the *upper confidence bounds for trees* (UCT) algorithm [7], which is regarded as the main representative of the field. UCT optimally balances the exploitation and exploration of actions in each state and does not require domain-specific knowledge. Despite this, its level of play is relatively low in practice when not aided by additional enhancements. In order to increase the playing strength, researchers often use expert knowledge or domain-specific heuristics, which decreases the generality of such algorithms. Improving the performance, while retaining the generality and scalability, has proven difficult [2].

The UCT algorithm selects actions according to a weighted exploratory term and a state value that is statistically averaged by *Monte Carlo* sampling. This algorithm, and also other MCTS algorithms in general, can be classified as *reinforcement learning* [8] methods. Research in this field developed the *temporal difference* (TD) learning paradigm [9], which combines the advantages of Monte Carlo sampling and dynamic programming. TD learning is capable of bootstrapping – i.e., updating the state value estimates without waiting for the final feedback. This is why it often outperforms learning by plain Monte Carlo sampling and may improve the performance of UCT as well, without sacrificing generality.

We introduce an approach that uses TD learning – instead of the default Monte Carlo statistical average – to estimate the values of nodes in the MCTS tree. We focus on the TD(λ) [8] algorithm, because it naturally integrates into the MCTS backpropagation step and tree structure. Three variants of the new algorithm are presented; one of them has been empirically designed to be compatible with a slightly modified variant of the *all moves as first* (AMAF) enhancement [10]. Benchmarks on the game of Gomoku asses the performance, scalability, and sensitivity to parameter values of the new algorithm in comparison to several UCT-based algorithms. Additionally, we evaluate its compatibility with AMAF and the *rapid action value estimation* (RAVE) enhancement, which is highly effective in Go [11]. Experiments on four different games measure the general performance of the new algorithm.

In the next section we briefly overview the research done on combining TD learning with MCTS and explain how our approach relates to reward-weighting methods [2]. Section III gives the background from which we derived our new algorithm. In Section IV we describe the integration of TD(λ) into UCT and in Section V its combination with the AMAF heuristic. The benchmarks are presented in Section VI. Section VII concludes with an overview of our work and gives suggestions for future research.

II. Related Work

The idea of combining the advantages of TD learning [9] and MCTS [1] is well-known. Silver *et al.* [12] define the *temporal-difference search* method that uses TD to learn the weights of a heuristic state evaluation function for Go. They successfully merge it with the alpha-beta search, but unsuccessfully with MCTS. In contrast, our research is focused on directly evaluating the nodes in the MCTS tree by TD learning, independently of the domain features or current state.

In the scope of reinforcement learning (RL) [8], Osaki *et al.* [13] developed a method that takes advantage of Monte Carlo simulations to gather winning probabilities as rewards in non-terminal positions. Their TDMC(λ) algorithm outperforms plain TD(λ) [9] in Othello. Although they do not use the MCTS framework, they propose further research to replace Monte Carlo simulations with the UCT [7] algorithm. Our idea is inverse – we start with UCT and integrate TD learning in its

978-1-4799-3547-5/14/1.00 ©2014 IEEE

framework. We enhance the backpropagation step to calculate the values by TD learning and memorize them in the tree structure, where they are later exploited by the tree policy.

The discounting mechanics from RL decrease a reward's weight proportionally with the distance from the state it was received. In the context of MCTS, this translates to weighting the iterations according to their duration, e.g., diminishing the weight of rewards with increasing distance to game end. Xie and Liu [14] explore this idea by modifying the backpropagation step – they partition the sequence of MCTS iterations and weight differently each partition. Their results show that later partitions deserve a higher weight, since they generally contain shorter iterations, which are more accurate. Kocsis and Szepesvari [7] diminish the exploration bias in their UCT algorithm when descending the tree; however, such weighting disregards that nodes at the same tree level may have different distances to the terminal position. Therefore, the rewards are not differentiated according to the game duration. In comparison with the described methods, our use of decaying eligibility traces from TD(λ) does not generalize across iterations or tree levels, but is more specific in the way that it applies to each reward and tree node independently.

Cowling *et al.* [15] also experiment with a reward-weighting scheme similar to the decaying eligibility traces. They exponentially decrease the final reward based on the number of plies from the root to the terminal state with the discount parameter $0 < \lambda < 1$. Their implementation updates all nodes by the same value – this differs from the TD approach that we use, where the reward decays also within the tree. Additionally, they test only a single value and in combination with specific UCT enhancements on the card game Magic: The Gathering. We provide a more detailed description in the broader framework of TD learning and present a more comprehensive benchmark.

III. Background

We briefly describe the TD(λ) and UCT algorithms on which our method is based, and the AMAF and RAVE enhancements that we used in our experiments.

A. Temporal Difference (TD) Learning

The TD learning paradigm is the key achievement of decades of research in the field of reinforcement learning [8]. One of its main representatives is the TD(λ) algorithm [9]. TD(λ) updates a state value $V_{\mathrm{TD}}(s_t)$ according to the immediate reward r_{t+1} and the value of the following state in the next time step $V_{\mathrm{TD}}(s_{t+1})$. It uses the eligibility traces mechanism, which propagates updates back in time to nodes visited earlier. Each state is paired with a trace $e(s_t)$, which decays exponentially with the number of time steps since that state was visited. The decay rate is defined by the parameter λ. The state value update rule for a single time step is

$$V_{\mathrm{TD}}(s_t) \leftarrow\ V_{\mathrm{TD}}(s_t) + \alpha e(s_t)\delta_t\ , \tag{1}$$

where

$$\delta_t =\ r_{t+1} + \gamma V_{\mathrm{TD}}(s_{t+1}) - V_{\mathrm{TD}}(s_t) \tag{2}$$

is the *temporal difference error* for which each node will be adjusted according to a step rate α and its *eligibility trace*

$$e(s_t) \leftarrow \begin{cases} \gamma\lambda e(s_t) + 1, & \text{if visited at time } t; \\ \gamma\lambda e(s_t), & \text{otherwise.} \end{cases} \tag{3}$$

The discounting factor γ lowers the importance of future rewards. The parameters α, γ and λ should be in the range of $[0, 1]$. The algorithm's convergence rate is affected by the initial V_{TD} values, thus it is meaningful to adjust them according to the expected distribution of rewards (e.g, initial values of 0.5 for rewards in the range of $[0, 1]$).

B. Upper Confidence Bounds for Trees (UCT)

The UCT algorithm [7] is one of the first and most widely used MCTS methods. It implements the four main MCTS iteration steps [1]: node *selection*, tree *expansion*, *playout* or *simulation* until terminal state, and *backpropagation* of the reward. The descent in the tree (i.e., the selection step) is guided by a *tree policy* that evaluates nodes based on the bandit algorithm UCB1 [16]. It selects children nodes with the highest value of

$$Q_{\mathrm{UCT}} = Q_{\mathrm{MC}} + c_{n_{\mathrm{p}},n}\ , \tag{4}$$

where

$$Q_{\mathrm{MC}} = \frac{\sum R_i}{n} \tag{5}$$

is the average reward received from a node (i.e., state) until the game end and

$$c_{n_{\mathrm{p}},n} = 2C_p\sqrt{\frac{2\log n_{\mathrm{p}}}{n}} \tag{6}$$

is the exploration bias, which is defined by the number of visits n of a node, the number of visits of its parent node n_{p}, and a weighting constant C_p. The rewards R_i are obtained from each MCTS iteration i by following a *default policy* (also called *playout policy*) that selects random actions in the simulation step. The process is repeated for an arbitrary number of iterations.

C. All Moves As First (AMAF)

The AMAF heuristic is often used in combination with MCTS algorithms in the game-playing domain. Its numerous variants and extensions proved effective in games with similar dynamics to the game of Go [2]. This approach keeps track of all actions that were selected during a playout and assigns them the reward as if they were selected in each visited node in the tree descent. Among numerous AMAF variants [2] we employ α-AMAF [17], which keeps a separate counter of visits n_{AMAF} and rewards R_{AMAF}. When applied to the UCT tree policy, the value of a node gets defined by

$$Q_{\alpha\text{-AMAF}} = \alpha\frac{R_{\mathrm{AMAF}}}{n_{\mathrm{AMAF}}} + (1-\alpha)Q_{\mathrm{UCT}}\ , \tag{7}$$

where α is a weighting parameter in the range of $[0, 1]$.

D. Rapid Action Value Estimation (RAVE)

The RAVE enhancement is an extension of AMAF where the weight α decreases with the number of node visits. Gelly and Silver [11] propose several schemes for changing the value of α; however, we use the simpler scheme by Helmbold and Parker-Wood [17], because it performed best on our benchmarks. We slightly modify it to allow the parameter V to hold a negative or zero value. This way the scheme defines the AMAF weight by

$$\alpha = \begin{cases} 0, & \text{for } V = 0; \\ \operatorname{sgn}(V) \cdot \max\left(0, \frac{|V|-n}{|V|}\right), & \text{otherwise,} \end{cases} \tag{8}$$

where parameter V sets the threshold for node visits n when the AMAF estimates will not be used anymore.

IV. Temporal Difference Values in UCT

We aim to improve general performance of the UCT algorithm [7]. In this section we present an enhancement for UCT that benefits from TD learning [9] to estimate state values. We extend the UCT tree policy defined by (4) to include TD values V_{TD} and weight them with an additional parameter w_{TD}. In this way, a node value is defined by

$$Q_{\text{TD-UCT}} = \; w_{\mathrm{TD}}\; V_{\mathrm{TD}} + (1 - w_{\mathrm{TD}})\; Q_{\mathrm{MC}} + c_{n_{\mathrm{p}},n}\;. \tag{9}$$

Such weighting scheme is well-known among other MCTS enhancements, e.g., in the AMAF enhancement [10]. It allows a linear combination of the two evaluators Q_{MC} and V_{TD}. The values V_{TD} are calculated in UCT's backpropagation step by one of the TD methods. We name this framework as the general *temporal difference values in UCT* (TD-UCT) algorithm.

We present three variants of TD-UCT that employ the TD(λ) algorithm [9] to calculate TD values V_{TD}. The variants differ in the number of additional parameters and in complexity. Two are simplified to only weight the rewards according to the distance from the game end, whereas one also bootstraps from previous estimates.

A. TD-UCT Single Backup

The temporal difference update rule by (2) requires the values of the states in the next time step $V_t(s_{t+1})$; however, these are unavailable in the playout of the UCT algorithm, since their estimates are not memorized in the tree structure. To solve this, we simplify the TD mechanics by assuming that those values have already converged to the value of the final reward R_i obtained from the playout. By this assumption, the states in the playout hold a value of $\gamma^{d_{\mathrm{E}}} R_i$, where d_{E} is the distance to the final state (i.e., number of plies until the game end). The first non-zero update occurs at the transition from the playout back to the tree leaf node. Its TD error is

$$\delta_1 = \gamma^P R_i - V_{\mathrm{TD}}(\mathrm{leaf})\;, \tag{10}$$

where P is the number of steps in the playout and $V_{\mathrm{TD}}(\mathrm{leaf})$ is the current TD value of the leaf node in the tree. If we set the TD-UCT algorithm to consider only this (first) update, and to omit all remaining updates within the tree (i.e., to omit bootstrapping), we get the update rule

$$V_{\mathrm{TD}}(s) \leftarrow V_{\mathrm{TD}}(s) + \alpha_{\mathrm{TD}}(\lambda\gamma)^{d_{\mathrm{L}}}\delta_1\;, \tag{11}$$

where d_{L} is the distance to the leaf node. This rule updates the values $V_{\mathrm{TD}}(s)$ of nodes occurring in the MCTS backpropagation step from the selected leaf up to the root. We name this as the *TD-UCT single backup* (SB) variant. It introduces four new parameters – w_{TD}, α_{TD}, γ and λ. In the context of MCTS, parameter γ weights a reward relatively to the number of plies in the entire iteration, whereas parameter λ weights it only for the length of the MCTS tree. If $0 < \gamma < 1$, then rewards from iterations with more plies are considered less reliable. We denote the step rate parameter as α_{TD} not to confuse it with the AMAF weighting parameter α. The iterative steps of TD-UCT SB are outlined in Algorithm 1. The tree policy implements the evaluation by (9), whereas the backpropagation implements the TD updates by (10) and (11). For a two-player scenario, the values R and δ must be negated between each backpropagation step up the tree. The playout procedure is enhanced to return the number of plies in the simulation.

B. TD-UCT Weighted Rewards

We simplify the TD-UCT SB variant to maximally reduce the number of parameters. We remove the leaf node bias $V_{\mathrm{TD}}(\mathrm{leaf})$ from (10), and set $\alpha_{\mathrm{TD}} = 1$ and $\lambda = 1$ in (11), and $w_{\mathrm{TD}} = 1$ in (9). The latter causes the temporal difference value V_{TD} to fully replace the average reward Q_{MC}. The tree policy and backpropagation equations shorten to

$$\begin{aligned} Q_{\text{TD-UCT}} &= \; V_{\mathrm{TD}} + 2C_p\sqrt{\frac{2\log n_{\mathrm{p}}}{n}}\;, \\ V_{\mathrm{TD}}(s) &\leftarrow \; V_{\mathrm{TD}}(s) + \gamma^{d_{\mathrm{T}}} R_i\;, \end{aligned} \tag{12}$$

where $d_{\mathrm{T}} = d_{\mathrm{L}} + P$ is the number of steps to the terminal state. The resulting algorithm uses a simple weighted sum of rewards in conjunction with the UCT's exploratory bias; hence, we name it *TD-UCT weighted rewards* (WR). This variant introduces only two new parameters, w_{TD} and γ, at the cost of omitting part of UCT's and TD learning's dynamics.

C. TD-UCT Merged Bootstrapping

Starting from TD-UCT SB we empirically devised a more complex variant that performs well in combination with the AMAF enhancement. It includes bootstrapping, i.e., it updates a state value according to the value of the next state. At the end of the MCTS playout, each visited node is updated by

$$\begin{aligned} V_{\mathrm{TD}}(s_t) &\leftarrow \; V_{\mathrm{TD}}(s_t) + \alpha_{\mathrm{TD}}(\lambda\gamma)^{d_{\mathrm{L}}}\gamma^P\delta_t\;, \\ \delta_t &= \; R_i + \gamma V_{\mathrm{TD}}(s_{t+1}) - V_{\mathrm{TD}}(s_t)\;. \end{aligned} \tag{13}$$

Here $V_{\mathrm{TD}}(s_{t+1})$ is the value of the children node that was selected by the tree policy during the selection step. We

Algorithm 1. The iterative process of TD-UCT SB.

```
1: procedure TD-UCT-ITERATION(rootNode)
2:     leafNode ← TREEPOLICY(rootNode)
3:     (R, P) ← SIMULATEPLAYOUT(leafNode)
4:     BACKPROPAGATE(leafNode, R, P)
5: end procedure

6: procedure TREEPOLICY(node)
7:     while node.state is not terminal do
8:         if every children of node visited at least once then
9:             node ← BESTCHILD(node)
10:        else
11:            node ← random unvisited child of node
12:            expand tree by adding node as leaf
13:            return node
14:        end if
15:    end while
16:    return node
17: end procedure

18: procedure BESTCHILD(node)
19:    evaluate each child of node by (9)
20:    return child with highest value (break ties randomly)
21: end procedure

22: procedure SIMULATEPLAYOUT(node)
23:    s ← node.state
24:    P = 0                                  ▷ initialize ply counter
25:    while s is not terminal do
26:        P ← P + 1
27:        s ← state following a random action from s
28:    end while
29:    get reward R from final state s
30:    return (R, P)
31: end procedure

32: procedure BACKPROPAGATE(node, R, P)
33:    δ ← γ^P · R − node.V_TD                ▷ the TD error by (10)
34:    e ← 1                                  ▷ eligibility trace
35:    while node is not null do
36:        node.visits ← node.visits + 1
37:        node.rewards ← node.rewards + R
38:        node.V_TD ← node.V_TD + α_TD · e · δ
39:        e ← e · λ · γ
40:        node ← parent of node
41:    end while
42: end procedure
```

assume that the state after a terminal node does not exist – it has a neutral value. This update rule merges two types of backups: one is the same as used in TD-UCT SB by (10) and (11), whereas the other is a bootstrapping backup. The latter decreases its step rate exponentially with $\gamma\lambda$ and does not backpropagate intermediate backups to previous states – it does not use eligibility traces. We name this variant as *TD-UCT merged bootstrapping* (MB). Its parameters have the same effect as in TD-UCT SB, with the addition that γ here influences also the rate of bootstrapping and not only the weights of rewards.

V. Combining TD-UCT With AMAF

In our experiments we evaluate the TD-UCT algorithm in combination with two variants of the AMAF heuristic. One variant is the original α-AMAF that we presented in Section III-C. In this case we combine α-AMAF and TD-UCT by replacing Q_{UCT} in (7) with $Q_{\text{TD-UCT}}$ from (9). The second variant is a slightly modified α-AMAF that we present below.

The α-AMAF2 Modification

We change the original α-AMAF to merge UCT and AMAF counters before the tree policy evaluates a node by (4). This modification calculates the weighted sums of visits and rewards for each state by

$$\begin{aligned} R_{\alpha\text{-AMAF2}} &= \alpha\ R_{\mathrm{AMAF}} + (1-\alpha)R\ , \\ n_{\alpha\text{-AMAF2}} &= \alpha\ n_{\mathrm{AMAF}} + (1-\alpha)n\ , \end{aligned} \tag{14}$$

and then uses these values in (5) and (6). Contrary to α-AMAF, here the number of AMAF visits affects UCT's exploratory bias. Furthermore, through the same bias, now the parent's AMAF visit count joins the equation. We refer to this variant as the α-AMAF2 enhancement and the RAVE2 enhancement when RAVE is employed to calculate the weight α. Although this is a simple modification, to our knowledge it has not been described in the literature yet.

VI. Experimental Results

To assess the performance of our methods, we compare the playing strength of several algorithms based on UCT on two-player games in different benchmark scenarios. The evaluated algorithms always play as second player. The playing strength is expressed as the winning rate against the first player, which uses plain UCT with $C_p = 0.2$. The winning rate is given in percentage, where a draw counts as 0.5 points.

To achieve the best performance of each algorithm, we optimized the learning parameters with a linear reward-penalty learning automata [18]. This was done offline on 10,000 to 100,000 games, depending on the benchmark and the algorithm. Details about the optimization process are given in each of the following subsections.

The set of evaluated algorithms includes plain UCT, three variants of TD-UCT presented in Section IV, and their combinations with α-AMAF and RAVE enhancements described in Sections III and V.

A. Benchmarks on Gomoku 7x7 at 100 Iterations

Most of our experiments were based on Gomoku (also called Five in a Row). In this game, two players alternate in placing pieces on empty intersections on the playing board. The first player that achieves five pieces of his own color in a row (vertically, horizontally or diagonally) wins. This is a very simplified variant of the game of Go. This basic version of Gomoku was solved up to boards of size 15x15. Additional rules have been proposed to lower the advantage of the first player; however, our implementation does not use them.

The main benchmark scenario consisted of a Gomoku board size of 7x7 with each player computing 100 MCTS

TABLE I. PERFORMANCE AGAINST UCT WITH $C_p = 0.2$ ON GOMOKU 7X7. BOTH PLAYERS COMPUTE 100 MCTS ITERATIONS PER MOVE. EVALUATED ALGORITHMS PLAY AS SECOND PLAYER.

UCT variant	Win Rate [%]	TD-UCT variant	Win Rate [%]
Plain UCT	55.2	WR	72.4
α-AMAF	62.0	SB	74.7
RAVE2	64.2	MB	75.1
RAVE	69.7	MB with α-AMAF	75.7
α-AMAF2	70.1	MB with RAVE2	79.5
		SB with α-AMAF2	80.4
		MB with α-AMAF2	88.7

Results averaged from one million repeats; 95% confidence interval is below 0.1%.

iterations per move (Table I). Prior to the evaluation, we optimized the parameter values on the same setting. Due to this optimization, the evaluated plain UCT player achieves a win rate higher than 50% against the first player, which uses non-optimized UCT.

Our TD-UCT enhancement increases the playing strength of UCT considerably – all variants achieve a performance slightly higher than UCT with AMAF or RAVE enhancements. The TD-UCT variants receive an additional boost in performance in conjunction with α-AMAF2 and RAVE2, but not in conjunction with the original variants. We evaluated all combinations of TD-UCT variants SB and MB with α-AMAF, α-AMAF2, RAVE and RAVE2; however, only those presented in Table I increase the performance. TD-UCT MB with α-AMAF2 performs substantially better than with RAVE2 and achieves the best performance of all the evaluated algorithms. The gain of AMAF and RAVE enhancements is greater in the TD-UCT MB variant than in the SB variant. We noticed that the major increase in performance is due to the bootstrapping term in TD-UCT MB and the presence of α-AMAF2 visit counters in the UCT's exploratory bias. Apart from TD-UCT, the α-AMAF2 variant exhibits a better performance than the original α-AMAF also in conjunction with plain UCT – it achieves a win rate comparable to RAVE.

B. Scalability

In the following Gomoku benchmarks we evaluate the sensitivity to parameter value optimization and scalability of four algorithms from Table I: plain UCT, UCT with RAVE, TD-UCT MB, and TD-UCT MB with α-AMAF2. The presented results were averaged from 20,000 to 100,000 repeats, so that the 95% confidence interval is below 1%.

1) Stronger Opponent: We evaluate the algorithms against an opponent that performs a different (lower or higher) number of MCTS iterations per move (Fig. 1). The performance ranking is nearly equal as in Table I. By increasing the opponent's strength the algorithms' win rates decrease; however, TD-UCT with α-AMAF2 is least affected and increases its relative advantage over the others. The evaluated players used parameters obtained from offline optimization against an opponent with 100 iterations per move. Optimization to a stronger opponent improved the performance only slightly. The ranking does not change even when playing against an opponent enhanced by RAVE or TD-UCT.

2) Higher Number of Iterations: All evaluated algorithms suffer from optimization overfitting to a fixed number of

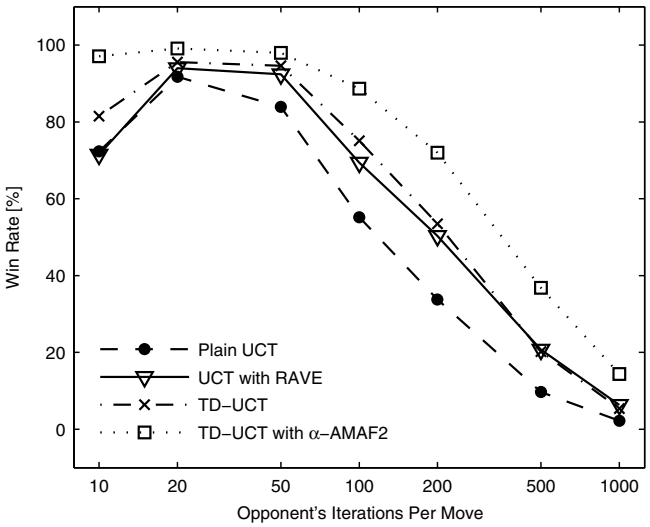
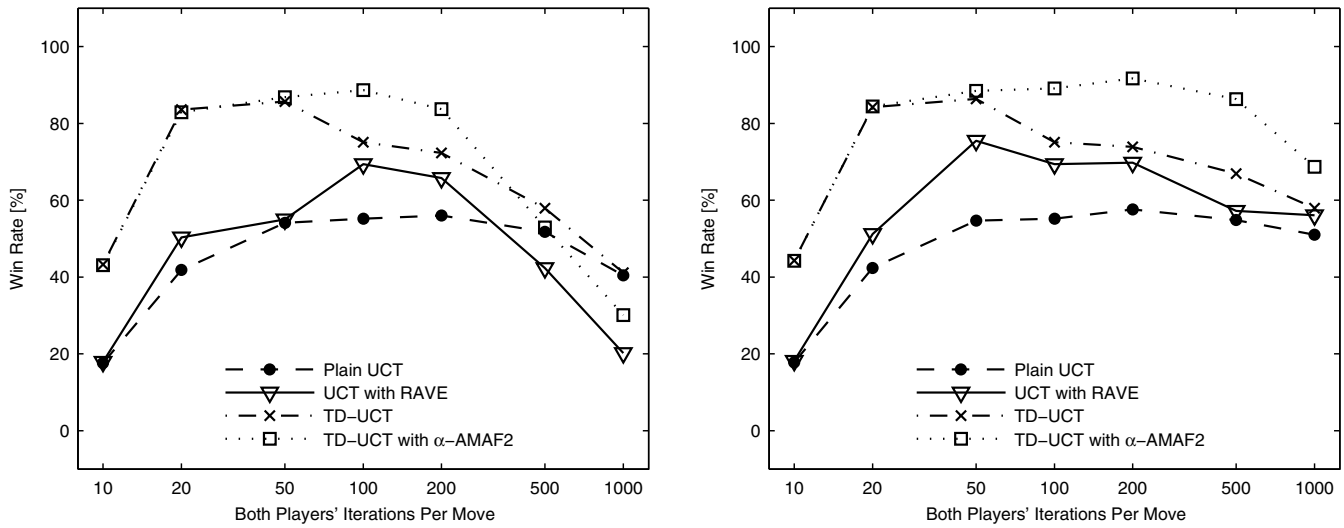


Fig. 1. Performance against UCT on Gomoku 7x7 when increasing the opponent's strength (i.e., the number of MCTS iterations per move). Evaluated players compute 100 iterations per move. The 95% confidence intervals are below 1%.

iterations per move (Fig. 2a). The least sensitive is plain UCT, followed by TD-UCT without AMAF. At 1,000 iterations per move both perform better than in combination with AMAF or RAVE enhancements – we infer that these are more sensitive to parameter value optimization. Despite this, both TD-UCT algorithms outperform UCT with RAVE at every setting; the difference is largest at a lower number of iterations.

The evaluated algorithms reach their best performance if optimized for each iteration setting individually (Fig. 2b). In comparison to the plain UCT, the advantage of other algorithms decreases with the increasing number of iterations per move. At 1,000 iterations, both UCT with RAVE and TD-UCT reach a performance only slightly higher than the plain UCT. Still TD-UCT with α-AMAF2 performs best; the gain over plain UCT is up to four-times higher than RAVE's at the highest number of iterations. To ensure fair results, all algorithms were optimized offline for an equal amount of time (i.e., number of games). For TD-UCT we optimized only the parameters C_p and w_{TD}, with the addition of α when α-AMAF2 was used. The remaining parameters α_{TD}, γ and λ had the same values as in previous benchmarks. When optimizing all TD-UCT's parameters, we achieved an additional 1-3% increase in win rate (not shown on figure).

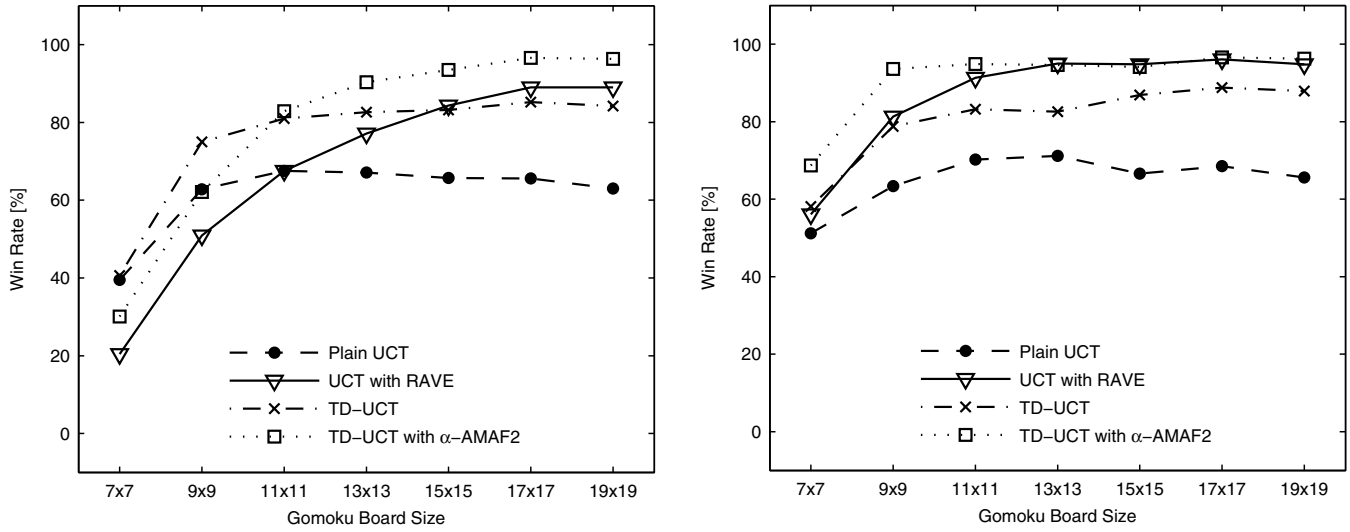
3) Larger State Space: When comparing the performance on a larger Gomoku board, we notice that TD-UCT with α-AMAF2 outperforms UCT with RAVE on all settings where the parameters were not optimized before the evaluation (Fig. 3a). This benchmark additionally confirms that α-AMAF and RAVE are more sensitive to parameter optimization, since both plain UCT and TD-UCT perform better at smaller board sizes. Consequently, the performance increase due to optimization is most notable for algorithms enhanced with AMAF or RAVE (Fig. 3b). In this case, both TD-UCT with α-AMAF2 and UCT with RAVE perform equally well at larger board sizes. When increasing the board size, the performance gap between these two algorithms decreases;



(a) Optimized at 100 iterations.

(b) Optimized for each setting.

Fig. 2. Performance against UCT on Gomoku 7x7 when increasing the number of MCTS iterations per move of both players. Comparison between offline optimization on 100 iterations (*left figure*) and offline optimization for each number of iterations individually (*right figure*). The 95% confidence intervals are below 1%.



(a) Optimized on size 7x7 at 100 iterations.

(b) Optimized for each setting.

Fig. 3. Performance against UCT on Gomoku when increasing the board size. Both players compute 1,000 iterations per move. Comparison between offline optimization on 100 iterations on a 7x7 board (*left figure*) and offline optimization on 1,000 iterations for each board size individually (*right figure*). The 95% confidence intervals are below 1%.

however, their advantage over plain UCT increases. The higher the ratio between search space size and number of iterations per move, the higher is the benefit of enhanced algorithms over plain UCT. These enhancements behave as quick value estimators for less explored areas of the state space.

C. Computation Time

We measured the computation time on Gomoku 7x7 by observing the number of simulated games per second. The TD-UCT algorithms run equally fast as the UCT algorithm, whereas both AMAF and RAVE enhancements run $10-15\%$ longer. However, the optimization of TD-UCT may take longer, due to its higher number of parameters. The program code was written in C++ and not parallelized. Experiments ran on a single core of an Intel Core i7 CPU.

D. Performance on Different Games

To verify the generality of our approach, we measured the winning rates on a set of games. Table II shows the results for 10 settings on four different games: Hex, Connect

TABLE II. PERFORMANCE AGAINST UCT WITH $C_p = 0.2$ AT DIFFERENT GAMES AND SETTINGS. THE FOUR EVALUATED PLAYERS WERE OPTIMIZED OFFLINE FOR EACH GAME SETTING. EVALUATED ALGORITHMS PLAY AS SECOND PLAYER.

	Hex 7x7		Connect Four		Gomoku 9x9		Gomoku 19x19		Tic Tac Toe	
Evaluated Players' Iterations	100	1000	100	1000	1000	2000	1000	2000	10	100
Opponent Player's Iterations	100	1000	100	1000	2000	4000	4000	8000	10	100
MCTS Algorithms' Win Rates [%]										
Plain UCT	57.8	49.8	44.4	47.2	37.1	30.3	42.6	50.9	33.3	36.1
UCT with RAVE	58.9	53.5	46.0	47.2	58.7	32.6	82.1	77.6	34.5	36.1
TD-UCT	72.9	55.7	53.7	48.9	49.2	39.7	64.6	62.1	42.1	37.2
TD-UCT with α-AMAF2	95.4	74.4	53.8	48.9	70.0	50.6	85.7	88.4	43.0	37.7
95% confidence interval	$\pm$0.7	$\pm$1.0	$\pm$0.1	$\pm$1.0	$\pm$1.0	$\pm$1.9	$\pm$1.4	$\pm$1.5	$\pm$0.4	$\pm$0.3

Four, Gomoku, and Tic Tac Toe. The evaluated algorithms were optimized offline for each setting against a plain UCT opponent with $C_p = 0.2$ and with an equal number of iterations per move. We ran less repeats of benchmarks with long computation time; for this reason the 95% confidence intervals differ among individual settings.

1) Hex 7x7: In this game players alternate in placing pieces on empty places on a rhombus board with a hexagonal grid. The winner is the first player to form a connected path of own pieces between two opposing borders of his color. The game cannot end in a draw. It was solved for board sizes up to 8x8. The first player has an advantage, so a pie rule is generally used after the first move; however, we do not implement it.

On Hex our TD-UCT algorithm with α-AMAF2 performs exceptionally well – the difference in win rates with other algorithms is greatest of all benchmarks. Even at a higher number of iterations it notably outperforms UCT with RAVE.

2) Connect Four: The players alternate in dropping pieces from the top of a board that is seven column wide and six rows high. They may choose any column that is not filled up. The winning player is the first who connects at least four own pieces in a straight line. The game has been solved with the outcome that the first player can force a win with perfect play.

Because this game is relatively simple, both players play well enough to finish most matches in a draw, even at a lower number of iterations per move. An UCT player with as much as 5,000 iterations could not elevate the win rate above 50% against an UCT player with 100 iterations (not shown in table). Nevertheless, TD-UCT manages to surpass this barrier with 100 iterations per move. On the contrary, both AMAF and RAVE prove ineffective. A reason may be that this is the only game in our benchmark set where the number of available actions is lower than the number of empty board places.

3) Gomoku 9x9 and 19x19: The description of this game was given in Section VI-A. We extend the benchmarks presented there to Gomoku boards of sizes 9x9 and 19x19.

The TD-UCT algorithm with α-AMAF2 performs best on all four settings. The second best is RAVE-enhanced UCT as it outperforms TD-UCT without AMAF on three settings where the ratio of search space size against the number of iterations is larger. TD-UCT without AMAF still performs better than plain UCT – also on Gomoku 9x9 at a higher number of iterations where RAVE gains little on the win rate.

The optimization on Gomoku 19x19 at a higher number of iterations is difficult and time consuming regardless of the evaluated MCTS algorithm. Even when optimizing plain UCT, the search algorithm struggled to find a satisfactory solution in a reasonable amount of time (i.e., a few days of computation). Moreover, the TD-UCT variants achieved a performance worse than plain UCT if not optimized to the higher number of iterations. This considered, UCT with RAVE proved most reliable regarding parameter sensitivity and optimization on Gomoku 19x19, despite reaching a slightly lower performance.

4) Tic Tac Toe: This is a very simple game with dynamics the same as Gomoku. It is played on a 3x3 board with a winning condition of three pieces in a straight line. Although the first player has an advantage, when both play optimally the game always ends in a draw.

Because of the small search space of this game, we evaluated the algorithms on a lower number of iterations per move, otherwise all matches would end in a draw. The performance gains of different enhancements are similar to that in Connect Four – AMAF and RAVE are mostly ineffective, whereas TD-UCT provides an increase on both settings.

VII. CONCLUSION

We proposed a new approach for including state values learned by temporal difference [9] into the general Monte Carlo tree search (MCTS) framework [1]. We introduced an enhancement of the upper confidence bounds for trees (UCT) algorithm [7] with the TD(λ) algorithm and named it *temporal difference values in UCT* (TD-UCT). Of the three presented TD-UCT variants, two weight the rewards according to the distance from the final state, whereas one also bootstraps from previous estimates. In addition to this, we presented a modification of the α-AMAF enhancement [17] that performs well with TD-UCT algorithms. A comparative evaluation of several UCT and TD-UCT variants assessed the performance of the new algorithm. We measured the win rates, scalability, and sensitivity to parameter values on four games: Gomoku, Hex, Connect Four, and Tic Tac Toe. We also examined the compatibility of these algorithms in combination with the all moves as first (AMAF) [10] and rapid action value estimation (RAVE) [11] enhancements.

The results show that our method generally increases performance of UCT on all benchmark games without prolonging the computation time. Moreover, it achieves an even higher win rate when combined with our modification of α-AMAF, although combining it with the original α-AMAF or RAVE enhancements was ineffective. Lastly, it seems equally robust to parameter values as the basic UCT algorithm, and is less

sensitive than AMAF and RAVE. These improvements come at the cost of up to four additional learning parameters, which make the search for their optimal values more difficult. Despite this, even the simplest TD-UCT variant noticeably outperforms UCT by introducing only one new parameter. Furthermore, the more complex variants can offer high performance even if some of their parameters are only loosely optimized.

Future experiments should evaluate our method on more challenging games, such as Go or Hex at a larger board size. Its overall level of play encourages its application also to General Game Playing [4]. Since TD-UCT efficiently merges with the AMAF heuristic, it is worth exploring combinations with other MCTS tree enhancements, such as progressive bias and progressive widening [19], and MCTS playout enhancements, such as simulation balancing [20].

TD-UCT could be easily implemented using transpositions [21] so that a directed acyclic graph is built instead of a tree. In this case the SARSA(λ) [22] algorithm may be more suitable than TD(λ) as it operates with the same equations, but evaluates state-action pairs instead of states. The values would be stored in the state-action records of the transposition table.

As a side note, the basic UCT algorithm enhanced with our modification of α-AMAF achieved the same level of play at Gomoku as with the original RAVE enhancement. This calls for additional investigation and a stronger empirical evaluation of the proposed modification.

The performance of our TD-UCT algorithm encourages further research in the general direction of integrating temporal difference learning into the MCTS framework. It additionally proves that weighting the rewards relatively to the duration of iterations is beneficial on a wide range of games. Apart from this, the role of bootstrapping and its contribution to the increase in performance is not yet clear. A better theoretical understanding of its dynamics, alongside exploration of the ideas presented in this section, may develop a new class of MCTS enhancements.

REFERENCES

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Proceedings of the 5th international conference on Computers and games*, ser. CG'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 72–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=1777826.1777833

[2] 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 Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[3] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud, "The grand challenge of computer Go: Monte Carlo tree search and extensions," *Communications of the ACM*, vol. 55, no. 3, pp. 106–113, 2012. [Online]. Available: http://doi.acm.org/10.1145/2093548.2093574

[4] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing," in *Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, ser. AAAI'08. AAAI Press, 2008, pp. 259–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=1619995.1620038

[5] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo Tree Search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, Dec. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5551182

[6] R. J. Lorentz, "Amazons Discover Monte-Carlo," in *Proceedings of the 6th international conference on Computers and Games*, ser. Lecture Notes in Computer Science, H. Herik, X. Xu, Z. Ma, and M. Winands, Eds. Springer Berlin Heidelberg, 2008, vol. 5131, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87608-3_2

[7] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Proceedings of the Seventeenth European Conference on Machine Learning*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Berlin/Heidelberg, Germany: Springer, 2006, pp. 282–293. [Online]. Available: http://www.sztaki.hu/ szcsaba/papers/ecml06.pdf

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[9] R. S. Sutton, "Learning to predict by the methods of temporal differences," in *Machine Learning*. Kluwer Academic Publishers, 1988, pp. 9–44.

[10] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proceedings of the 24th international conference on Machine learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 273–280. [Online]. Available: http://doi.acm.org/10.1145/1273496.1273531

[11] ——, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, Jul. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2011.03.007

[12] D. Silver, R. S. Sutton, and M. Müller, "Temporal-difference search in computer Go," *Machine Learning*, vol. 87, no. 2, pp. 183–219, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10994-012-5280-0

[13] Y. Osaki, K. Shibahara, Y. Tajima, and Y. Kotani, "An Othello evaluation function based on Temporal Difference Learning using probability of winning," *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 205–211, Dec. 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5035641

[14] F. Xie and Z. Liu, "Backpropagation Modification in Monte-Carlo Game Tree Search," in *Proceedings of the Third International Symposium on Intelligent Information Technology Application - Volume 02*, ser. IITA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 125–128. [Online]. Available: http://dx.doi.org/10.1109/IITA.2009.331

[15] P. I. Cowling, C. D. Ward, and E. J. Powley, "Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 4, pp. 241–257, Dec. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6218176

[16] 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. [Online]. Available: http://dx.doi.org/10.1023/A:1013689704352

[17] D. Helmbold and A. Parker-Wood, "All-Moves-As-First Heuristics in Monte-Carlo Go." *IC-AI*, pp. 605–610, 2009. [Online]. Available: http://www.soe.ucsc.edu/ dph/mypubs/AMAFpaperWithRef.pdf

[18] K. S. Narendra and M. A. L. Thathachar, *Learning automata - an introduction*. Prentice Hall, 1989.

[19] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive Strategies For Monte-Carlo Tree Search," *New Mathematics and Natural Computation*, vol. 4, no. 03, pp. 343–357, 2008. [Online]. Available: http://www.personeel.unimaas.nl/m-winands/documents/pmcts.pdf

[20] D. Silver and G. Tesauro, "Monte-Carlo simulation balancing," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. New York, NY, USA: ACM, 2009, pp. 945–952. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553495

[21] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in Monte Carlo tree search," *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 389–395, Dec. 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5035667

[22] G. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, 1994, no. September. [Online]. Available: ftp://ftp-svr.eng.cam.ac.uk/pub/pub/reports/auto-pdf/rummery_tr166.pdf

View publication stats