Accelerated UCT and Its Application to Two-Player Games

Junichi Hashimoto[1], Akihiro Kishimoto[2], Kazuki Yoshizoe[3], and Kokolo Ikeda[1]

[1] Japan Advanced Institute of Science and Technology
[2] Tokyo Institute of Technology and Japan Science and Technology Agency
[3] The University of Tokyo

Abstract. Monte-Carlo Tree Search (MCTS) is a very successful approach for improving the performance of game-playing programs. This paper presents the Accelerated UCT algorithm, which overcomes a weakness of MCTS caused by deceptive structures which often appear in game tree search. It consists in using a new backup operator that assigns higher weights to recently visited actions, and lower weights to actions that have not been visited for a long time. Results in Othello, Havannah and Go show that Accelerated UCT is not only more effective than previous approaches but also improves the strength of FUEGO, which is one of the best computer Go programs.

1 Introduction

MCTS has achieved the most remarkable success in developing strong computer Go programs [6, 7, 10], since traditional minimax-based algorithms do not work well due to a difficulty in accurately evaluating positions. The Upper Confidence bound applied to Trees (UCT) algorithm [13] is representative of MCTS. It is applied to not only Go but also other games such as Amazons [12, 15] and Havannah [19].

MCTS consists of two procedures, the Monte-Carlo simulation called *playout*, and tree search. In a playout at position P, each player keeps playing a randomly selected move until reaching a terminal position where the outcome of the terminal position (e.g., win, loss or draw) o is defined by the rule of the game. The outcome of the playout at P is defined as o. In the tree-search procedure, each move contains a value indicating the importance of selecting that move. For example, UCT uses the Upper Confidence Bound (UCB) value [1] (explained later) as such a criterion.

MCTS repeats the following steps until it is time to play a move. First, starting at the root node, MCTS traverses the current tree in a best-first manner by selecting the most promising move until reaching a leaf node. Then, if the number of visits reaches a pre-determined threshold, the leaf is expanded to build a larger tree. Next, MCTS performs a playout at the leaf to calculate its outcome. Finally, MCTS traces back along the path from the leaf to the root and update the values of all the affected moves based on the playout result at the leaf.

At each internal node n, UCT selects move j with the largest UCB value defined as:

$$\mathrm{ucb}_j := r_j + C\sqrt{\frac{\log s}{n_j}}, \tag{1}$$

where r_j is the winning ratio of move j, s is the number of visits to n, n_j is the number of visits to j, and C is a constant value that is empirically determined.

The inaccuracy of the winning ratio is measured by the second term, the *bias term*, which decreases as the number of playouts increases. Given infinite number of playouts, selecting the move with the highest winning ratio is proved to be optimal. However, the transitory period in which UCT chooses the suboptimal move(s) can last for a very long time [4] due to the over-optimistic behavior of UCT.

One way to overcome the above situation is to discount wins and losses for the playouts performed previously. Because the current playout is performed at a more valuable leaf than previously preformed ones, the current playout result is considered to be more reliable. Thus, MCTS can search a more important part of the search tree by forgetting the side effects of previous playouts. Although Kocsis and Szepesvári's Discounted UCB algorithm [14] is an example of such an approach, results for adapting it to tree search have not been reported yet except for an unsuccessful report in the Computer Go Mailing List [8].

This paper introduces a different way of focusing on more recent winning ratios than Discounted UCB. Its contributions are:

1. Development of the *Accelerated UCT* algorithm that maintains the reliability of past winning ratios and focuses on exploring the subtrees where playout results are recently backed up.
2. Experimental results showing the potential of Accelerated UCT over plain UCT and Discounted UCT, which is an application of Discounted UCB to tree search, in Othello, Havannah and Go.
3. Experimental results clearly showing that Accelerated UCT with Rapid Action Value Estimate [9, 20] further improves the strength of FUEGO [7], a strong computer Go program that is freely available.

The structure of the paper is as follows: Section 2 explains a drawback of UCT. Section 3 reviews the literature. Section 4 describes Accelerated UCT, followed by experimental results in Section 5. Section 6 discusses conclusions and future work.

2 Drawback of UCT

Kocsis and Szepesvári proved that the UCB value converges on the optimal value [13]. This indicates that UCT can eventually select an identical move to the minimax (i.e., optimal) strategy. However, while this theoretical property assumes the condition of which unlimited time is given to UCT, UCT must determine the next move to play in *limited time*, that is, the number of playouts UCT can perform in practice is not usually large enough to converge on the optimal value. As discussed in details

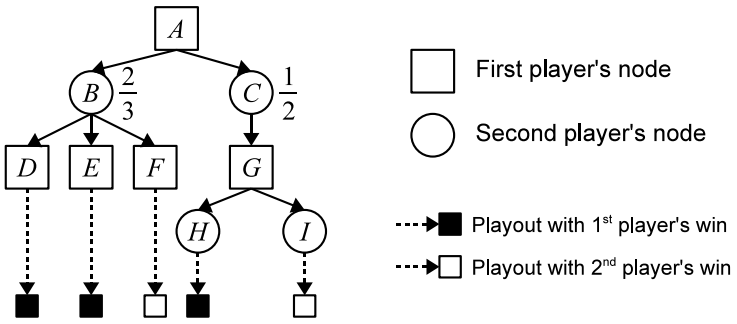

Fig. 1. An example of a deceptive structure which leads UCT to select a losing move even if playout results are accurate. The winning ratio of the first player is shown.

below, UCT therefore suffers from a *deceptive structure* in its partially built tree, which may be a cause of selecting a wrong move that may lead to losing a game.

Because UCT's playouts often mistakenly evaluate winning positions as losses or vice versa, one example of deceptive structures is caused by an inaccurate winning ratio computed by such faulty playout results. Playout policies could be modified to decrease the frequency of encountering the deceptive structures with knowledge-based patterns (e.g., [5, 10]).

However, even if playout results are accurate, UCT may still suffer from deceptive structures in the currently built tree. This drawback of UCT is explained with the help of Fig. 1. Assume that D, E and H are winning positions and F and I are losing positions for the first player. Then, $A \rightarrow B$ and $A \rightarrow C$ are losing and winning moves with the optimal strategy, respectively, because $A \rightarrow B \rightarrow F$ is a loss and $A \rightarrow C \rightarrow G \rightarrow H$ is a win. Also assume that UCT has currently performed only one playout at each leaf in the tree built as in this figure[4]. If the playout results at D, E and H are wins and the playout results at F and I are losses, the winning ratio of $A \rightarrow B$ is larger than that of $A \rightarrow C$ (2/3 versus 1/2), resulting in UCT choosing a losing move. UCT can obviously calculate the more accurate winning ratio by visiting B more frequently. However, UCT remains to be *deceived* to select $A \rightarrow B$ as a more promising move than $A \rightarrow C$ until $A \rightarrow B$ turns out to be valueless.

In general, deceptive structures tends to appear when MCTS must select the best move at a node with only a few promising children (e.g., ladders in Go).

[4] For simplicity, we also assume that C in equation 1 is very small (close to 0), although the drawback of UCT occurs even with large C.

3 Related Work

A number of approaches have been presented to correct the deceived behavior of MCTS in the literature, including modified playout policies (e.g., [5, 10, 11, 17]). Although modifying playout policies is presumed to decrease deceptive structures in the search tree, these techniques are mostly based on domain-dependent knowledge. Additionally, this approach cannot completely remove the deceptive structures as shown in Fig. 1. In contrast, we aim at avoiding the deceptive structures in a domain-independent way.

The rest of this section deals with related work based on different formulas than UCB to bypass such deceptions. These approaches are orthogonal to modifying playout policies and can be usually combined with these policies.

Coquelin and Munos showed an example in which UCT works poorly due to its over-optimistic behavior if it performs only a small number of playouts [4]. They presented the Bandit Algorithm for Smooth Trees (BAST) to overcome this issue by modifying the bias term of the UCB formula. A theoretical property was proved about the regret bound and the playout sizes performed on suboptimal branches, when their smoothness assumption is satisfied in the tree. Since they did not show empirical results in games, it is still an open question whether BAST is effective for two player games or not.

Discounted UCB [14] gradually forgets past playout results to value more recent playout results. It introduces the notion of *decay*, which was traditionally presented in temporal difference learning [18]. The original UCB value is modified to the Discounted UCB value as explained below.

In the multi-armed bandit problem, let $r^D_{t,j}$ and $n^D_{t,j}$ be the discounted winning ratio of branch j and the discounted visits to j, respectively, after the t-th playout is performed. The Discounted UCB value $\mathrm{ducb}_{t,j}$ is defined as[5]:

$$\mathrm{ducb}_{t,j} := r^D_{t,j} + C\sqrt{\frac{\log s^D_t}{n^D_{t,j}}}, \tag{2}$$

where $s^D_t := \sum_i n^D_{t,i}$ and C is a constant. Discounted UCB incrementally updates $r^D_{t,j}$ and $n^D_{t,j}$ in the following way:

$$n^{\mathrm{D}}_{t+1,j} \leftarrow \lambda \cdot n^{\mathrm{D}}_{t,j} + \mathbb{I}(t+1,j), \tag{3}$$

$$r^{\mathrm{D}}_{t+1,j} \leftarrow \left(\lambda \cdot n^{\mathrm{D}}_{t,j} \cdot r_{t,j} + \mathbb{R}(t+1,j)\right) / \left(\lambda \cdot n^{\mathrm{D}}_{t,j} + \mathbb{I}(t+1,j)\right), \tag{4}$$

where λ is a constant value of decay ranging $(0, 1]$, $\mathbb{I}(t, j)$ is set to 1 if j is selected at the t-th playout or 0 otherwise. $\mathbb{R}(t, j)$ is defined as the result of the t-th playout (0 or 1 in this paper for the sake of simplicity) at j if j is selected, or 0 if j is not selected. We assume $r^D_{0,j} = n^D_{0,j} = 0$

[5] Precisely, Kocsis and Szepesvári used the UCB1-Tuned formula [10] to define the Discounted UCB algorithm. However, we use the standard UCB1 formula here, since we believe that this difference does not affect the properties of Discounted UCB.

for any j but $\mathrm{ducb}_{0,j}$ has a very large value so that j can be selected at least once. Note that Discounted UCB is identical to UCB if $\lambda = 1.0$. Additionally, selecting the right λ plays an important role in Discounted UCB's performance.
Discounted UCB selects branch j with the largest discounted UCB value and performs a playout at j, and then updates the discounted UCB values of *all* the branches. In other words, while Discounted UCB updates $\mathrm{ducb}_{t,j}$ for selected branch j, the Discounted UCB values for the other unselected branches are also discounted. Discounted UCB repeats these steps until it performs many playouts enough to select the best branch.
While Discounted UCB could be applied to tree search in a similar manner to UCT, one issue must be addressed (see Subsection 5.1 for details) and no success in combining Discounted UCB with tree search has been reported yet.
Ramanujan and Selman's UCTMAX_H combines UCT and minimax tree search [16]. It simply replaces performing a playout with calling an evaluation function at each leaf and backs up its minimax value instead of the mean value. Although they showed that UCTMAX_H outperforms UCT and minimax search in the game of Mancala, their approach is currently limited to domains where both minimax search and UCT perform well.
Other related work includes approaches using other algorithms as baselines rather than UCT and their applicability to UCT remains an open question. For example, instead of computing the winning ratio, Coulom introduced the "Mix" operator that is a linear combination of robust max and mean [6].

4 Accelerated UCT

Our Accelerated UCT algorithm aims at accelerating to search in a direction for avoiding deceptive structures in the partially built tree. Similarly to Discounted UCB, Accelerated UCT considers recent playouts to be more valuable. However, unlike the decay of Discounted UCB, Accelerated UCT non-uniformly decreases the reliability of subtrees that contain the past playout results.
As in UCT, Accelerated UCT keeps selecting the move with the largest *Accelerated UCB value* from the root until reaching a leaf. The Accelerated UCB value aucb_j for move j is defined as:

$$\mathrm{aucb}_j := r_j^{\mathrm{A}} + C\sqrt{\frac{\log s}{n_j}}, \tag{5}$$

where the bias term is identical to UCB and r_j^{A} is the *accelerated winning ratio* (explained later) defined by the notion of *velocity*. If Accelerated UCT is currently at position P for the t-th time, the velocity $v_{t,j}$ of *each of* legal moves j at P is updated in the following way:

$$v_{t+1,j} \leftarrow v_{t,j} \cdot \lambda + \mathbb{I}(t+1, j), \tag{6}$$

where $\mathbb{I}(t+1, j)$ is 1 if move j is selected at P and is 0 otherwise, and λ is a decay ranging $(0, 1]$, which has a similar effect to the decay of

Discounted UCB and is empirically preset. For any move j, $v_{0,j}$ is set to 0.

Let r_i^{A} be the accelerated winning ratio of move i that creates position P and r_j^{A} be the accelerated winning ratio of move j at P. When Accelerated UCT backs up a playout result, it updates r_i^{A} with the following velocity-based weighted sum of r_j^{A}:

$$r_i^{\mathrm{A}} := \sum_{j \in LM(P)} w_j \cdot r_j^{\mathrm{A}}, \tag{7}$$

where $LM(P)$ is all the legal moves at P and $w_j := v_{t,j}/(\sum_{k \in LM(P)} v_{t,k})$. If move j is selected, $v_{t,j}$ and w_j are increased, resulting in giving r_j^{A} a heavier weight. Additionally, Accelerated UCT is identical to UCT if $\lambda = 1$.

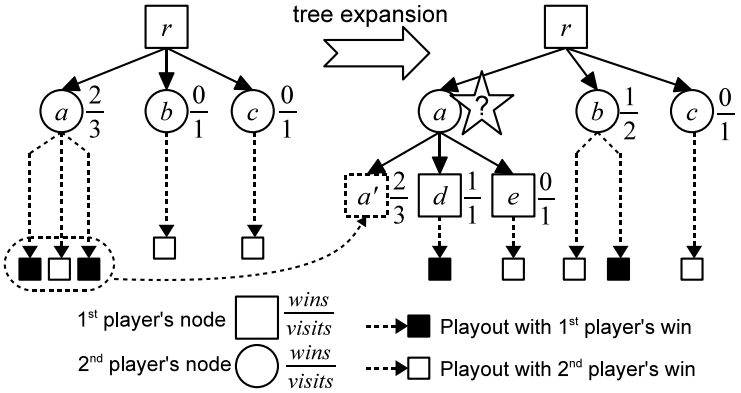

Fig. 2. An example illustrating the necessity of virtual child in Accelerated UCT

When a leaf is expanded after performing several playouts, Accelerated UCT must consider a way of reusing these playout results. We prepare an additional child called *virtual child* for this purpose.

Fig. 2 illustrates the virtual child of node a, represented as a'. Nodes a, b and c are leaves in the left figure. Assume that three playouts are performed at a and the winning ratio of move $r \to a$ is 2/3. Then, if a is expanded, Accelerated UCT generates two *real* children (i.e., d and e) and one virtual child a' as shown in the right figure. Additionally, assume that one playout per real child is performed (the playout results are a win at d and a loss at e).

In case of plain UCT, the winning ratio of $r \to a$ can be easily calculated as 3/5 in this situation, because the playout results previously performed at a (i.e.,the winning ratio of 2/3) are accumulated at $r \to a$. However, Accelerated UCB updates the accelerated winning ratio $r_{r \to a}^{\mathrm{A}}$ of $r \to a$

based on the winning ratios of $a \to d$ and $a \to e$ in the original definition. These moves do not include the value of 2/3. We therefore add one virtual child a' and (virtual) move $a \to a'$ to set the winning ratio of $a \to a'$ to 2/3. That is, as if there were three moves ($a \to a'$, $a \to d$ and $a \to e$) at a, Accelerated UCT updates $r^{\mathrm{A}}_{r \to a}$.

Note that the velocity of a' is also considered but is always decayed because a' is never selected.

5 Experiments

5.1 Implementation Details

We implemented the plain UCT, Discounted UCT (Discounted UCB plus tree search), and Accelerated UCT algorithms to evaluate the performance of these algorithms in the games of Othello, Havannah and Go. All the algorithms were implemented on top of FUEGO 0.4.1[6] in Go. In contrast, we implemented them from scratch in Othello and Havannah.

Since Discounted UCB updates all the branches of the root in the multi-armed bandit problem, one possible Discounted UCT implementation is to update all the Discounted UCB values in the currently built tree. However, because this approach obviously incurs non-negligible overhead, our implementation updates the Discount UCB values in the same way as Accelerated UCT updates velocities. In this way, our Discounted UCT implementation can recompute the Discounted UCB values of "important" moves with a small overhead.

5.2 Setup

Experiments were performed on a dual hexa-core Xeon X5680 machine with 24 GB memory. While this machine has 12 cores in total, we used a single core to run each algorithm with sufficient memory.

We set a limit of the playout size to 50,000 when each algorithm determined the move to play. Although Discounted/Accelerated UCT requires an extra overhead to compute Discounted/Accelerated UCB values compared to plain UCT, we observed that this overhead was negligible.

We held a 1000-game match to compute the winning percentage for each algorithm in each domain. A draw was considered to be a half win when the winning percentage was calculated. The ratio of draws to the total number of games ranged 0.9–2.2% in Havannah, while this number was at most 7.8% in Othello. The games results were always either wins or losses in 9×9 Go with 7.5 komi. Because MCTS has randomness for playout results, we disabled the opening book for the experiments and thus obtained a variety of games per match.

Table 1. Performance comparison

(a) Accelerated vs plain UCT

	Winning percentage (%)		
k	Othello	Havannah	Go
1	39.2 ± 3.1	24.2 ± 2.7	0.8 ± 0.6
2	52.5 ± 3.2	25.1 ± 2.7	41.3 ± 3.1
3	$\mathbf{56.3 \pm 3.2}$	47.8 ± 3.2	54.5 ± 3.1
4	52.7 ± 3.1	$\mathbf{58.2 \pm 3.1}$	$\mathbf{56.0 \pm 3.1}$
5	51.3 ± 3.2	50.6 ± 3.2	53.0 ± 3.2
6	47.4 ± 3.2	48.0 ± 3.2	51.0 ± 3.2
7	48.2 ± 3.2	50.2 ± 3.2	50.1 ± 3.2

(b) Discounted vs plain UCT

	Winning percentage (%)		
k	Othello	Havannah	Go
1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
2	1.3 ± 0.7	0.0 ± 0.0	0.0 ± 0.0
3	28.0 ± 2.8	8.0 ± 1.7	0.0 ± 0.0
4	47.5 ± 3.2	31.3 ± 2.9	18.6 ± 2.5
5	49.0 ± 3.2	48.5 ± 3.2	45.8 ± 3.2
6	$\mathbf{49.2 \pm 3.2}$	$\mathbf{52.0 \pm 3.2}$	46.6 ± 3.2
7	48.2 ± 3.2	50.0 ± 3.2	$\mathbf{48.7 \pm 3.2}$

5.3 Performance Comparison of the Plain, Discounted, and Accelerated UCT Algorithms

Table 1 shows the winning percentages of Accelerated/Discounted UCT with various decays against plain UCT in Othello, Havannah and 9×9 Go with 95% confidence intervals, calculated by $2\sqrt{p(100-p)/1000}$, where p is the winning percentage. The best results are marked by bold text. We varied decay $\lambda = 1 - 0.1^k$ $(1 \leq k \leq 7)$ for both Discounted and Accelerated UCT. For example, $\lambda = 0.9999999$ with $k = 7$ and $\lambda = 0.9$ with $k = 1$. This implies that λ becomes close to 1 more extremely with larger k, resulting in Discounted/Accelerated UCT behaving more similarly to plain UCT.

FUEGO's default policy that performs a smarter playout based on game-dependent knowledge was used in the experiments in Go. Additionally other important enhancements except Rapid Action Value Estimate (RAVE)[7] [9] were turned on there. In contrast, in Havannah and Othello, when a playout was performed, one of the legal moves was selected with uniform randomness without incorporating any domain-specific knowledge. Additionally no techniques enhancing the performance of UCT variants were incorporated there. C was set to 1.0 in all the domains.

The winning percentages in the table indicate that Accelerated UCT was consistently better than Discounted UCT. There is at least one case of k where Accelerated UCT was statistically superior to plain UCT in each domain, although the best k depends on the domain. In contrast, even with the best k, Discounted UCT was at most as strong as plain UCT, implying the importance of introducing a different way of decaying the winning ratio than the Discounted UCB value.

[6] The source code is available at `http://fuego.sourceforge.net/`. The latest implementation is version 1.1. However, version 0.4.1 was the latest one when we started implementing the aforementioned algorithms.

[7] We intend to show the potential of the Accelerated and Discounted UCT algorithms against UCT in this subsection. See the performance comparison with turning on RAVE in the next subsection.

The constant value of C may impact the performance of Discounted UCT since the best C might be different from plain and Accelerated UCT due to the different formula of the biased term in Discounted UCT. However, we did not exploit the best C for plain and Accelerated UCT either. Moreover, we verified that the values of the biased terms of Discounted and Accelerated UCT with the best λ were very similar when we ran these algorithms with many positions.
Despite inclusions of FUEGO's essential enhancements to improve its playing strength except RAVE, Accelerated UCT still achieved non-negligible improvement. Additionally even if no enhancements were incorporated in Othello and Havannah, Accelerated UCT was better than plain UCT. These results indicate that Accelerated UCT was able to remedy the deceived behavior of UCT which could not be corrected completely by the enhancements presented in the previous literature (e.g., modifications to playout policies).
If the winning ratio was over-discounted (i.e., in case of small k), both Discounted and Accelerated UCT performed poorly. However, Accelerated UCT was still more robust than Discounted UCT to the change of λ (see Table 1 again). In the extreme case of of $k = 1$ where $\lambda = 0.9$, we observed that Discounted UCT won no games against plain UCT. This result indicates that Discounted UCT suffers from undesirable side effects if it eventually forgets all the past playout results that often contain useful information. In contrast, Accelerated UCT often bypasses this drawback of Discounted UCT, because the backup rule of Accelerated UCT still takes into account the past valuable playout results.

5.4 Performance Comparison with RAVE in Go

The RAVE enhancement [9] plays a crucial role in drastically improving the strength of many computer Go programs including FUEGO. One question is how to combine Discounted or Accelerated UCT with RAVE. This subsection answers the question and shows experimental results when RAVE is turned on in FUEGO, that is, we used the best configuration of FUEGO as a baseline.
RAVE assumes that there is a strong correlation between the result of a playout and the moves that appear during performing that playout as in [2, 3]. RAVE then sets the playout result as the value of these moves (we call this value the *RAVE playout value*) so that the UCB values of the moves (called the *RAVE values* precisely) can be updated with their RAVE playout values even at different positions. While the original RAVE formula appears in [9], FUEGO uses a slightly different formula in [20]. The RAVE value of move j (rave_j) is defined as[8]:

$$\text{rave}_j := \frac{n_j}{n_j + W_j} r_j + \frac{W_j}{n_j + W_j} r_j^{\text{RAVE}} \tag{8}$$

[8] The RAVE value could have a bias term as in the UCB value. However, it is omitted in many computer Go programs in practice because the second term of rave_j often has a similar effect to the bias term. The bias term was not therefore included in the experiments here since FUEGO also performs best with no bias term.

where r_j is the winning ratio of j, n_j is the number of visits to j, r_j^{RAVE} is the RAVE playout value of j, and W_j is the unnormalized weight of the RAVE estimator (see [20] for details). Instead of using the UCB value, FUEGO keeps selecting move j with the highest rave_j from the root until reaching a leaf to perform a playout.

RAVE tries to empirically converge the value more quickly than UCT, which is successful in current Go programs. As a result, this property might have a complementary effect on avoiding deceptive structures in the tree.

In our Discounted UCT implementation, r_j and n_j in rave_j are replaced by $r_{t,j}^{\mathrm{D}}$ and $n_{t,j}^{\mathrm{D}}$ in Equations 3 and 4, respectively[9]. In contrast, in our Accelerated UCT implementation, only r_j is replaced by r_j^{A} in Equation 7.

Table 2. Performance comparison with FUEGO with switching on RAVE and all important enhancements in Go

(a) Accelerated vs FUEGO

k	Winning percentage (%)
1	51.2 ± 3.2
2	51.9 ± 3.3
3	54.5 ± 3.4
4	$\mathbf{55.9 \pm 3.5}$
5	53.8 ± 3.4
6	54.6 ± 3.4

(b) Discounted vs FUEGO

k	Winning percentage (%)
1	0.0 ± 0.0
2	0.0 ± 0.0
3	4.4 ± 0.3
4	41.6 ± 2.6
5	47.9 ± 3.0
6	$\mathbf{49.8 \pm 3.1}$

Table 2 shows the winning percentages of Discounted and Accelerated UCT with RAVE against FUEGO with the best configuration which also includes RAVE. Accelerated UCT statistically performs better than FUEGO, which implies that RAVE does not always fix the problem of deceptions in MCTS and Accelerated UCT may correct some of the deceived behaviors of MCTS. In the best case, the winning percentage of Accelerated UCT against FUEGO was 55.9 % if λ is set to 0.9999. In contrast, Discounted UCT was again at most as strong as FUEGO, as we saw similar tendencies in the previous subsection. Discounted UCT lost all the games if the winning ratio is over-discounted with small k (i.e., $k \leq 2$), while Accelerated UCT was very robust to the change of k. Again, this indicates that Discounted UCT inherently has a side effect of forgetting past valuable playout results.

[9] Unlike in the definition of Discounted UCB, t and j indicate the situation after the tth update *for move* j is performed.

6 Concluding Remarks

We have developed the Accelerated UCT algorithm that avoids some of the deceptions that appear in the search tree of MCTS. Our experimental results have shown that Accelerated UCT not only outperformed plain and Discounted UCT in a variety of games but also contributed to improving the playing strength of FUEGO, which is one of the best computer Go programs.
Although Accelerated UCT is shown to be promising, the most important future work is to develop a technique that automatically finds a reasonable value of decay λ. At present, we must try to find a good value of λ empirically by hand. In our experiments, the best λ depends on the target domain. Additionally, since we believe that the best λ also depends on the time limit, it would be necessary for Accelerated UCT to automatically change the value of λ, based on a few factors such as the shape of the current search tree.

Acknowledgements. This research was supported by the JST PRESTO program. We thank Tomoyuki Kaneko and Martin Müller for their valuable comments on the paper.

References

1. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002.
2. B. Bouzy and B. Helmstetter. Monte Carlo Go developments. In *Proc. of the 10th International Conference on Advances in Computer Games, (ACG10)*, volume 263 of *International Federation for Information Processing*, pages 159–174. Kluwer Academic, 2003.
3. B. Brügmann. Monte Carlo Go. 1993. `http://www.ideanest.com/vegos/MonteCarloGo.pdf`.
4. P.-A. Coquelin and R. Munos. Bandit algorithms for tree search. In *Proc. of the 23rd Conference on Uncertainty in Artificial Intelligence, (UAI2007)*, pages 67–74. AUAI press, 2007.
5. R. Coulom. Computing Elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, 2007.
6. R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proc. of the 5th International Conference on Computers and Games, (CG2006)*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2007.
7. M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.
8. S. Gelly. Discounted UCB. posted to Computer Go Mailing List, 2007. `http://www.mail-archive.com/computer-go@computer-go.org/msg02124.html`.

9. S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *Proc. of the 24th International Conference on Machine Learning, (ICML2007)*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280, 2007.
10. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report RR-6062, INRIA, 2006.
11. S.-C. Huang, R. Coulom, and S.-S. Lin. Monte-Carlo simulation balancing in practice. In *Proc. of the 7th International Conference on Computers and Games, (CG2010)*, volume 6515 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2010.
12. J. Kloetzer, H. Iida, and B. Bouzy. A comparative study of solvers in Amazons endgames. In *Proc. of the IEEE Symposium on Computational Intelligence and Games, (CIG2008)*, pages 378–384. IEEE Press, 2008.
13. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Proc. of the 17th European Conference on Machine Learning, (ECML2006)*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.
14. L. Kocsis and C. Szepesvári. Discounted UCB. Video Lecture, 2006. In the lectures of PASCAL Second Challenges Workshop 2006, Slides are available at `http://www.lri.fr/~sebag/Slides/Venice/Kocsis.pdf`. Video is available at `http://videolectures.net/pcw06_venice/`.
15. R. Lorentz. Amazons discover Monte-Carlo. In *Proc. of the 6th International Conference on Computers and Games, (CG2008)*, volume 4630 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2008.
16. R. Ramanujan and B. Selman. Trade-offs in sampling-based adversarial planning. In *Proc. of 21st International Conference on Automated Planning and Scheduling, (ICAPS2011)*, pages 202–209. AAAI, 2011.
17. D. Silver and G. Tesauro. Monte-Carlo simulation balancing. In *Proc. of the 26th International Conference on Machine Learning, (ICML2009)*, volume 382 of *ACM International Conference Proceeding Series*, pages 945–952, 2009.
18. R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.
19. F. Teytaud and O. Teytaud. Creating an upper-confidence-tree program for Havannah. In *Proc. of the International Conference on 12th Advances in Computer Games, (ACG12)*, volume 6048 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2009.
20. D. Tom and M. Müller. A study of UCT and its enhancements in an artificial game. In *Proc. of the 12th International Conference on Advances in Computer Games, (ACG12)*, volume 6048 of *Lecture Notes in Computer Science*, pages 55–64. Springer, 2009.